Spatial Database Modeling and Consistency in Web Frameworks Engineering:
the case of python-powered web2py framework

Habtom B Tsega
February, 2010
Spatial Database Modeling and Consistency in Web Frameworks Engineering: the case of python-powered web2py framework

by

Habtom B Tsega

Thesis submitted to the International Institute for Geo-information Science and Earth Observation in partial fulfillment of the requirements for the degree in Master of Science in GeoInformatics.

Degree Assessment Board

Thesis advisor
Dr. Javier Morales
Drs. B.J. Barend Köbben

Thesis examiners
Chair: Dr.Ir. R.A. de By
External examiner: Dr. Theodor Förster

international institute for geo-information science and earth observation
enschede, the netherlands
Disclaimer

This document describes work undertaken as part of a programme of study at the International Institute for Geo-information Science and Earth Observation (ITC). All views and opinions expressed therein remain the sole responsibility of the author, and do not necessarily represent those of the institute.
Abstract

No doubt, almost all spatial applications rely on the use and manipulation of geographically-aware data, re-usable if and only if stored and managed appropriately. In standalone and remote applications, this is quite prudently possible by making use of the most effective and as-old-as-computing-itself database technology. Stored data if not safe-guarded from illegal actions that invalidate their very consistency (hence, existence) would obviously become shortly out of use. This explains the need for the employment of consistency monitoring scheme by applying Database Management System (DBMS) at the back-end.

On the contrary, the state-of-the-art Web application development tools, with Web frameworks in the lead, rely on a very trivial methods of consistency verification mechanisms that are confined almost totally on the client-side. They employ database back-ends to only store their information content. The task of shielding the data from any instance of inconsistency is given to very few and limitedly effective pre-defined methods on the application-layer. The question to be raised here is, why databases only for data storage, while they can benefit the Web application with a lot more functionality, including comprehensive consistency control?

The first portion of this project outlines a method to bring Web frameworks into the realm of spatial applications, as most of them are originally meant only for non-spatial applications. What was outlined has been implemented on Web2py, one out of the hundreds of such open-source software frameworks. Equally important, the second portion of the project attempts to solve the limitedness of consistency checking paradigm being used by Web frameworks today in spatial and non-spatial contexts. It proposes a well-designed database modeling approach that accommodates the specification of not only pre-defined but also user-defined application-dependent constraint rules, that guarantee the validity of the spatial database. To test the proposed architectural re-design, it was implemented on a working use-case, the Amazonia species distribution data management system.

Keywords

MVC, web2py, Spatial database, Consistency, Model-View-Controller, Web frameworks, GeoDjango, Spatial Web, UML, UWE, Web Modeling, WFS-T, PostGIS
Acknowledgements

Foremost, I would like to thank GOD for giving me an incredibly strong mom and everything I need.

Next, my heartfelt gratitude goes to: my first supervisor, Dr Javier Morales whose immeasurable guidance, contribution and assistance throughout the project period enabled me to develop an understanding of the subject matter and give my thesis a shape it currently has: my second supervisor Drs. B.J. Barend Köbben whose incredible fortitude, hard work and technical help made things possible, which were thoroughly impossible without his kind help: Dr.Ir. R.A. de By whose marvelous patience, friendly-approach and clarification of every question I pose anytime I need it, starting from the proposal till the end of the project work: Netherlands Fellowship Program (NFP) for giving me the golden chance to pursue my MSc study in GeoInformatics in a very beautiful country and people called Holland: Kibreab Hailemichael and Mrs Yibralem Kidane for being there to help me whenever I needed it: Professor Massimo Di Pierro, the creator of web2py, for inventing such a powerful, comprehensible and fun-to-play-with Web framework, plus for his very quick responses to all my technical difficulties related with web2py: last but not least, all friends (GFM and Home), family and whoever is reading this document.

Haftish21
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 The Web engineering</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2 The database technology</td>
<td>1</td>
</tr>
<tr>
<td>1.1.3 The Web database system</td>
<td>2</td>
</tr>
<tr>
<td>1.1.4 The Web frameworks engineering</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Problem definition</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Research Objectives</td>
<td>7</td>
</tr>
<tr>
<td>1.3.1 General objective</td>
<td>7</td>
</tr>
<tr>
<td>1.3.2 Specific objectives</td>
<td>7</td>
</tr>
<tr>
<td>1.4 Research questions</td>
<td>8</td>
</tr>
<tr>
<td>1.5 Innovation aimed at</td>
<td>8</td>
</tr>
<tr>
<td>1.6 Related prior work</td>
<td>8</td>
</tr>
<tr>
<td>1.7 Methodology adopted</td>
<td>9</td>
</tr>
<tr>
<td>1.8 Organization of the thesis</td>
<td>10</td>
</tr>
<tr>
<td><strong>2 Review of Web database Application (WDA) Modeling</strong></td>
<td>13</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Web-database application modeling process</td>
<td>13</td>
</tr>
<tr>
<td>2.2.1 Dimension-1: The levels</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2 Dimension-2: The phases</td>
<td>16</td>
</tr>
<tr>
<td>2.2.3 Dimension-3: The aspects</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Modeling tool</td>
<td>16</td>
</tr>
<tr>
<td>2.4 Model Driven Architecture (MDA)-compliant Web Engineering (WE) method for Web-Database Application (WDA) development process</td>
<td>18</td>
</tr>
<tr>
<td>2.5 The design-driven WDA modeling process</td>
<td>18</td>
</tr>
<tr>
<td>2.5.1 The Computational Independent Model (CIM): Requirements model</td>
<td>20</td>
</tr>
<tr>
<td>2.5.2 The Platform Independent Model (PIM): Conceptual and Logical models</td>
<td>21</td>
</tr>
<tr>
<td>2.5.3 The Platform Specific Model (PSM): Physical model</td>
<td>22</td>
</tr>
<tr>
<td>2.5.4 The Model-View-Controller (MVC) versus MDA</td>
<td>22</td>
</tr>
<tr>
<td>2.6 Conclusion</td>
<td>22</td>
</tr>
<tr>
<td><strong>3 Review of Spatial Database Consistency</strong></td>
<td>25</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>25</td>
</tr>
<tr>
<td>3.2 Consistency defined</td>
<td>25</td>
</tr>
<tr>
<td>3.3 Topological Model</td>
<td>26</td>
</tr>
<tr>
<td>3.4 Constraints and their Taxonomy</td>
<td>32</td>
</tr>
</tbody>
</table>
## Contents

- 3.4.1 Spatial constraints: based on spatial behavior ........................................ 32
- 3.4.2 Spatial constraints: based on level of abstraction .................................. 34
- 3.5 Constraint specification ............................................................................. 36
- 3.6 Constraints enforcement ............................................................................ 36
- 3.6.1 Preferred constraint implementation approaches .................................... 38
- 3.6.2 Constraints violation and response action .............................................. 39
- 3.7 Architectural re-engineering ...................................................................... 39
- 3.8 Introducing the PostgreSQL Relational DBMS (extended with PostGIS) .... 40
- 3.9 Conclusion ............................................................................................... 41

4 Spatialization of the Web Frameworks .......................................................... 43
- 4.1 Introduction ............................................................................................... 43
- 4.2 The spatial Web ......................................................................................... 43
  - 4.2.1 The Service-Oriented Architecture (SOA) triad ................................... 44
  - 4.2.2 OGC Web Services (OWS): OpenGIS® Consortium (OGC)® Web Services 45
- 4.3 Enhancing MVC spatially .......................................................................... 46
  - 4.3.1 Learning from pioneers ..................................................................... 46
  - 4.3.2 Extending the M (of MVC) in Web frameworks ................................. 49
  - 4.3.3 Extending the C (of MVC) in Web frameworks ................................. 53
  - 4.3.4 Extending the V (of MVC) in Web frameworks ................................. 56
- 4.4 Conclusion ............................................................................................... 56

5 The design process: design-driven modeling .................................................. 57
- 5.1 Introduction ............................................................................................... 57
- 5.2 Requirements Engineering: the CIM ......................................................... 57
  - 5.2.1 User requirements ............................................................................ 58
  - 5.2.2 Functional requirements ................................................................... 59
  - 5.2.3 Constraints identification ................................................................. 65
- 5.3 Content Model: PIM ................................................................................ 66
  - 5.3.1 Constraints specification ................................................................... 67
  - 5.3.2 Constraints enforcement ................................................................. 68
- 5.4 Conclusion ............................................................................................... 68

6 Implementation: Experiment the re-engineered method ................................ 69
- 6.1 Introduction ............................................................................................... 69
- 6.2 The M as a spatial database schema ........................................................... 69
  - 6.2.1 Web2py model ............................................................................... 70
  - 6.2.2 Populating the tables ....................................................................... 74
- 6.3 The V as a map interface .......................................................................... 74
  - 6.3.1 Setting-up spatial servers .................................................................. 74
  - 6.3.2 Utilizing OpenLayers Application Programming Interface (API) .... 75
  - 6.3.3 Cascading Style Sheets (CSS) ......................................................... 79
- 6.4 The C as a mediator ................................................................................ 79
- 6.5 Conclusion ............................................................................................... 81

7 Experimentation: Experiment the implementation .......................................... 83
- 7.1 Introduction ............................................................................................... 83
- 7.2 The facade ............................................................................................... 83
- 7.3 Playing with the controls ......................................................................... 84
  - 7.3.1 Layer Switcher ............................................................................... 84
  - 7.3.2 Filtering species distribution geometry .......................................... 85
  - 7.3.3 Zooming ......................................................................................... 85
  - 7.3.4 Map scale and mouth locator .......................................................... 85
  - 7.3.5 Navigation and history ..................................................................... 87
  - 7.3.6 Feature attribute pop-up ................................................................... 88
- 7.4 Data inserts and updates .......................................................................... 88
8 Summary, conclusions, recommendations and impact 95
8.1 Summary of the study ........................................... 95
8.2 Conclusions of the study ....................................... 96
8.3 Recommendations of the study ................................. 97
  8.3.1 Extend other constraint implementation options .......... 98
  8.3.2 Utilize OCL-based constraint specification ............... 98
  8.3.3 Make a full integration of the MDA ....................... 98
  8.3.4 Resolve performance problem ........................... 98
  8.3.5 Realize Spatial enrichments to Object-Constraint-Language (OCL) .... 98
  8.3.6 Enhance the current functionality of our Web Feature Service Transactional (WFS-T) prototype .... 99
  8.3.7 Utilize extended version of Unified Modeling Language (UML) for MVC modelling .... 99
  8.3.8 Put a powerful wrapper for the creation of useful database functions, callable from the Web later ......................... 99
  8.3.9 Comprise exception handlers inside Web frameworks ........ 99
  8.3.10 Make a full-fledged WDA design .......................... 100
  8.3.11 Intensify the current guideline for spatialization of Web frameworks .... 100
  8.3.12 Create a Object Relational Mapping (ORM)apper from Standard Query Language (SQL) into web2py M .................................................. 100
8.4 The Impact on other Web frameworks ......................... 100
  8.4.1 In terms of Modeling ...................................... 100
  8.4.2 In terms of Consistency .................................. 101
  8.4.3 In terms of Spatialization .............................. 101

Bibliography 103

A Extending the M of web2py 109
  A.1 Target geometric fields ..................................... 109
  A.2 The gField initializer ...................................... 110
  A.3 The gField creator ...................................... 110
  A.4 Wrapper for invoking server-side functions in web2py .... 111

B Constraints handlers 113
  B.1 Constructor of Checks in web2py ............................ 113
  B.2 Constructor of within trigger in web2py .................... 114
  B.3 Server-side functions ..................................... 115

C Phases of content model 123
  C.1 Logical database model ..................................... 123
  C.2 Physical database model ................................... 124
  C.3 Physical database model: edited version ................. 125
  C.4 Web2py model ........................................... 126
  C.5 Intermediary file ........................................ 127
Contents

D The V as a map interface 131
  D.1 Index file - Basic HTML 131
  D.2 Map definition 134
  D.3 Base Map 135
  D.4 Vector layer 135
  D.5 Controls and editing tools 138

E WFS-T Events (JavaScript) 141
  E.1 JavaScript functions 141

F The Controller file 149
  F.1 Controller actions 149
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Group of valid relationships[28]</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>Mapping between constraints specification and enforcement</td>
<td>68</td>
</tr>
<tr>
<td>5.2</td>
<td>Use-case constraints and their preferred options of enforcement</td>
<td>68</td>
</tr>
<tr>
<td>6.1</td>
<td>ORMapping - PostgreSQL/PostGIS to web2py</td>
<td>71</td>
</tr>
</tbody>
</table>
List of Tables

x
List of Figures

1.1 Classic Web-database application as three-tier server-client architecture ........................................ 3
1.2 MVC architecture ................................................................................................................................. 4
1.3 WDA architecture, a Web-framework involved .................................................................................. 5
1.4 Parallel database formation phases in database applications (design-driven) versus in web2py environment (implementation-driven) .......................................................... 6
1.5 Incompleteness of the set of consistency rules handled by Web frameworks ........................................ 6
1.6 Validators (highlighted) as consistency handlers in web2py Model (M of MVC) ................................... 6
1.7 How Web frameworks react to violations of constraints right from the Model (the role of Database Management System (DBMS) ignored) ........................................................................... 7
1.8 The method adopted, simplified ......................................................................................................... 11
2.1 The MDA ............................................................................................................................................ 14
2.2 The Dimensions of Web Applications modeling ................................................................................ 14
2.3 Hierarchy for metamodeling ............................................................................................................. 17
2.4 UWE-based MDA-compliant WDAs ................................................................................................. 19
2.5 Use-case Relationships ....................................................................................................................... 20
2.6 Hierarchical view of MVC ................................................................................................................. 23
3.1 Simplified internal structure of a topology ........................................................................................ 27
3.2 OGC® Simple Feature Hierarchical Model ..................................................................................... 28
3.3 Simplified illustration: Interior, closure and boundary of point, line and polygon features .......... 29
3.4 Simple topological relations of point feature with point, line and polygon features ................... 30
3.5 8 simple Polygon-Polygon topological relations ............................................................................. 30
3.6 33 simple Line-Line topological relations ........................................................................................ 31
3.7 19 simple Line-Polygon topological relations .................................................................................. 33
3.8 Constraints categorization ................................................................................................................. 33
3.9 Where are constraints applied? .......................................................................................................... 36
3.10 Proposed marriage between design-driven modeling and MVC architecture ............................... 40
4.1 SOA involving Publish-Find-Bind operations .................................................................................... 44
4.2 Relationship between clients/servers and some OGC® protocols .................................................. 46
4.3 Overall architecture of GeoDjango ................................................................................................... 47
4.4 Spatial extensions to non-spatial tables ............................................................................................. 48
5.1 Use case illustration ............................................................................................................................ 58
5.2 Activity diagram: depicts structure of add_distrib callable function ................................................ 60
5.3 Activity diagram: depicts structure of update_distrib callable function .......................................... 61
5.4 Activity diagram: depicts structure of delete_distrib callable function ......................................... 62
5.5 Activity diagram: depicts structure of restrict_distrib function ..................................................... 63
5.6 Activity diagram: depicts structure of add_species function ......................................................... 64
5.7 Activity diagram: depicts structure of add_family and add_genus functions ............................... 65
5.8 Conceptual database schema ............................................................................................................ 66
7.1 The facade of our web2py-powered CONSISTENT WFS-T prototype ........................................ 84
List of Figures

7.2 OpenLayers control tools: Layer Switcher ........................................... 85
7.3 Filtering example: distribution of Species 967 ....................................... 86
7.4 Unfiltered version of species distribution layer ....................................... 86
7.5 OpenLayers control tools: Zooming options ......................................... 87
7.6 OpenLayers control tools: Map scale and mouth locator .......................... 87
7.7 OpenLayers control tools: Map Overview window ................................... 88
7.8 OpenLayers control tools: feature info ............................................... 89
7.9 INSERTION of species distribution data ............................................... 89
7.10 UPDATING of species distribution data .............................................. 90
7.11 RESTRICT species distribution geometry to given list of interfluves .......... 91
7.12 DELETION of species distribution data .............................................. 92
7.13 INSERT of family data .................................................................. 92
7.14 INSERT of genus data .................................................................. 93
7.15 INSERT of invalid geometry data - outside legal bounding box ............. 94
7.16 Response given from the database as a result of an attempt to insert invalid geometry .......................... 94
C.1 Logical database model ................................................................. 123
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CIM</td>
<td>Computational Independent Model</td>
</tr>
<tr>
<td>CASE</td>
<td>Computer Assisted Software Engineering</td>
</tr>
<tr>
<td>CRUD</td>
<td>Create-Retrieve-Update-Delete</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>CSW</td>
<td>Web Catalog Services</td>
</tr>
<tr>
<td>DAL</td>
<td>Data Abstraction Layer</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>DDL</td>
<td>Data Definition Language</td>
</tr>
<tr>
<td>DEM</td>
<td>Digital Elevation Model</td>
</tr>
<tr>
<td>DML</td>
<td>Data Manipulation Language</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>EA</td>
<td>Enterprise Architect</td>
</tr>
<tr>
<td>FOL</td>
<td>First Order Logic</td>
</tr>
<tr>
<td>GDAL</td>
<td>Geospatial Data Abstraction Library</td>
</tr>
<tr>
<td>GeoIP</td>
<td>IP-based Geolocation library</td>
</tr>
<tr>
<td>GeoJSON</td>
<td>Geographic JavaScript Object Notation</td>
</tr>
<tr>
<td>GEOS</td>
<td>Geometry Engine, Open Source</td>
</tr>
<tr>
<td>GeoRSS</td>
<td>Geographic Really Simple Syndication</td>
</tr>
<tr>
<td>GIS</td>
<td>Geographic Information System</td>
</tr>
<tr>
<td>GiST</td>
<td>Generalized Search Tree</td>
</tr>
<tr>
<td>GML</td>
<td>Geographic Markup Language</td>
</tr>
<tr>
<td>HTML</td>
<td>Hyper Text Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>ISO</td>
<td>International Standards Organization</td>
</tr>
<tr>
<td>JTS</td>
<td>Java Topology Suite</td>
</tr>
<tr>
<td>JPEG</td>
<td>Joint Photographic Experts Group</td>
</tr>
<tr>
<td>JS</td>
<td>JavaScript</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>KML</td>
<td>Keyhole Markup Language</td>
</tr>
<tr>
<td>LBS</td>
<td>Location-based Services</td>
</tr>
<tr>
<td>MDA</td>
<td>Model Driven Architecture</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta-Object Facility</td>
</tr>
</tbody>
</table>
List of Figures

MVC       Model-View-Controller
OGC       OpenGIS® Consortium
OCL       Object-Constraint-Language
OMG       Object Management Group
ORM       Object Relational Mapping
OWS       OGC Web Services
PIM       Platform Independent Model
PNG       Portable Network Graphics
PSM       Platform Specific Model
qGIS      Quantum GIS
QVT       Query-View-Transformation
RE        Requirements Engineering
RDBMS     Relational Database Management System
RS        Remote Sensing
RSS       Really Simple Syndication
SDI       Spatial Data Infrastructure
SOA       Service-Oriented Architecture
SOAP      SOA communication Protocol
SRID      Spatial Reference system Identifier
SRS       Spatial Reference Systems
SQL       Standard Query Language
uDig      User-friendly Desk-top Internet GIS
UML       Unified Modeling Language
UWE       UML-based Web Engineering
WCS       Web Coverage Services
WDA       Web-Database Application
WE        Web Engineering
WFS       Web Feature Service
WFS-T     Web Feature Service Transactional
WGS       World Geodetic System
WKB       Well-Known Binary
WKT       Well Known Text
WMS       Web Map Services
WPS       Web Processing Services
WSDL      Web Services Description Language
XHTML     eXtensible Hyper Text Markup Language
XMI       XML Interchange Format
XML       eXtensible Mark-up Language
Chapter 1

Introduction

1.1 Motivation

1.1.1 The Web engineering

The Web is an implementation of the Internet that networks low-cost computer systems to build a structure for sharing and communicating local resources distributed all over the world, which makes it one among the distributed applications[24]. Many scholars argue and agree that ever since Tim Burners-Lee, a scientist at the CERN (European Center for Particle Physics), conceived a pragmatic notion of transforming the Internet into a medium of scientific documents sharing on August 6, 1991, the World Wide Web has become an imperative communication platform for all sorts of multimedia data (text, images, sound, video etc), executable code (script) and information transactions. Also, it revolutionarily transformed human computing capability and even personal lives[82]. It is progressively intruding into every aspect of human culture and growing in a speed that can be said exponential and unparalleled by any other technology[84].

In fact, the first-generation Web played only a publication role, holding not more than static Web pages. “The Web was originally envisaged as a mechanism for storing and providing access to documents”[24], and therefore was best labeled as static (also known as shallow or boiler-plate text) Web. The deep/dynamic Web followed with its possibility to vary its own presentation by generating contents on-the-fly from database repositories. Both Web eras share one common characteristic: they are read-only and less-interactive. Therefore, Web engineers categorized them as version 1.0 of the Web (Web 1.0). On the contrary, the second-generation Web, Wisdom Web (or Web 2.0), is founded under the innovative philosophy of ‘Web as a Read-Write medium’ and two-way user-centric interactivity[84]. Users are no more passive who only read what is published on the page, rather they can actively take part by reacting and putting their valuable inputs, such as in collaborative mapping. There follows some more batches of the Web which are far beyond the concern of the thesis and will not be considered here.

A Web application development is often wrongly looked at as a one-off process, which ends when the looked-for Web page is created and/or when a pilot usability study gives a sign of its effectiveness. However, more recently, Web Engineering is handling the subject very differently and professionally by an integrated use of “systematic and quantifiable approaches in order to accomplish the specification, implementation, operation, and maintenance of high quality Web applications”[48]. A Web application is no longer viewed as a build-it-and-they-will-come approach.

1.1.2 The database technology

Databases are seen as mimicry of human brains[82] in respect of their power that is technically termed as having Create-Retrieve-Update-Delete (CRUD) features: store, renew, interpret, take back and delete information about the reality. Whereas the limitedness of human brains to hold bulky data and to make speedy computational analysis has long been resolved by handing such tasks over to computers[82] (therefore, databases). Humankind is in a course of making quicker and more effective decisions today than in the pre-computer days.
1.1. Motivation

Martyn Prigmore in[77] described a database as “a persistent, self-describing, structured collection of related items of data designed for timely access of accurate information”. Such a system calls for a structure that poses a number of security, concurrency and complexity challenges of data treatment questions[83], especially for spatial applications that often involve capacious data and complex programs to process them. This comprehensive database management task is accommodated by a full-scale software tool known as DBMS. To communicate with DBMSs, SQL is the standard international language that any system is required to ‘speak’.

A Data model is often described to be an imperative component of database technology, and is therefore labeled as one of its most powerful building blocks. Databases abstractly imitate and mirror world realities using such carefully studied representation that captures required information and behavior about what is being modeled by means of set of mathematical concepts, operations and rules. The relational data model is the well-known and broadly used data model by most large commercial/enterprise and open DBMSs. It is based on an uncomplicated philosophy that data is stored inside interrelated tables (also referred to as relations) as rows (or tuples) and columns (or attributes). Building the model should be an initial phase in the development of database applications, hence it is not an ad-hoc stage, i.e., it seeks a formal design process which comes in three basic steps: conceptual, logical and physical, so that is labeled as design-driven. According to[24], a data model has at least three major components, relating to the way information is modeled as data:

- Structural component: a collection of structures and their types, a key to database formulation;
- Manipulative component: a collection of the operations applicable to the database;
- Integrity component: a set of rules/constraints that should be obeyed without any violation by the database for it to remain valid and consistent, so that consistency is enforced and/or preserved.

1.1.3 The Web database system

Databases are as old as computing itself[82] and are far older and more mature than the Web. “Databases often served several different applications, and so couldn’t be tailored to any particular one”[77]. However, the mutual interplay between databases and Web applications has always stayed to become the focal subject of engineering works within the sphere of information technology since the Web came onto the scene. In[24], this fact is indicated by making a clear distinction between two possible interactions: the Web-enhanced database system and the database-enhanced Web system. The latter is by far the arrangement that is needed to support the development of database-backed Web applications, which commonly are known to Web practitioners as WDAs, and will be the main concern of this study.

How can using a DBMS enhance a Web application? Webs rely on databases for greater dynamism and flexibility of their contents. The Web has its own persistent memory, more or less comparable to that in databases, which enables it to store hypermedia documents[24]. However, there are still a number of missing qualities to it as compared to the benefits gained from a database beneath it. Unlike storing hypermedia documents on the Web, database technology offers some extra benefits, as summarized below:

- The data stored in databases have clearly described and pre-documented structure and schema;
- Databases have no difficulty of handling large volumes of data;
- Databases support many data administration tasks - including maintenance of data integrity and multiple (concurrent) uses.

A Server, any service-provider program running on a host node, waits for a request to be initiated by the client, which is another program running on another node seeking access to the services offered by the service-provider node. (Node is meant here as any computer hardware and/or software). The server listens and serves the client by giving a relevant response to each request sent to it. This architecture is best designated as the client/server computing architecture. There are various types of server, among which the Web server is the one to implement the Web itself in two ways. Firstly, it on its own stores programs, information and services to be called by clients. Secondly, it provides a gateway to information stored on another third-party system, like a database. Furthermore, the database server is another type of server that runs the DBMS to manage the data traffic to and fro the database.
Likewise, Web-database applications encapsulate three-tier client/server computing architecture (see Figure 1.1)[105, 106, 46]. This is a structural design which embraces a Web-client (a browser) involving two servers: a Web server and a database server. The middle tier, the Web server (also known as Hypertext Transfer Protocol (HTTP) server) listens to requests sent from clients and responds by sending a replica of the requested static and dynamic data content back. To be able to perform its role, the Web server in turn acts like a client by requesting the DBMS to fetch raw, dynamically filtered or processed resources from the database server, for presentation and/or client-side processing. A network system executes the information exchange and interlinking of the nodes (and their peripherals). In no particular order, three components are the active players in this arrangement: the database, the Web and the network.

1.1.4 The Web frameworks engineering

All deep Web and wisdom Web applications supported by CRUD features and an underlying database back-end require dynamic and agile Web environments that go beyond a mere display of pages. Geographic (spatial) applications on the Web, commonly named Web-GISs, are among the foremost cases-in-point that entail highly dynamic and interactive Web settings. “Modern Web applications are full-fledged, complex software systems; therefore, the development of Web applications requires a methodologically sound engineering approach”[48]. “To successfully build complex Web-based systems and applications, both large and small, Web developers need to adopt a disciplined development of process and sound design methodologies and use better development tools”[84]. This is not a subject of the aesthetic value of the pages, designing visually charming Web facades, but the self-consistent and unfailing lifecycle of frequently changing data content at the back-stage of such an application[77].

Technology is a less pressing problem in this regard since many software tools exist in the information systems playground to support powerful Web developments. Among them are the software stacks that are conventionally christened by Web practitioners as ‘matured and full stack Web frameworks’, which are compilations of libraries that are helpful to easily and rapidly build and maintain agile Web applications[96]. They are often designed and developed collaboratively as social software. “The use of Web application frameworks can often reduce the number of errors in a program, both by making the code simpler, and by allowing one team to concentrate just on the framework”[97]. A Web framework is
1.2 Problem definition

There is therefore a software frame that is structured to ease the development of dynamic applications and services on the Web[96]. It is truly demanding to make an enumeration of existing Web frameworks. Better yet, “dynamically typed programming languages (especially ones with strong object-orientation) have been demonstrated to be both clean and efficient. Python with extremely clean syntax seems to be a natural alternative”[13]. Django, Pylons, TurboGears, web2py, Grok and Zope are the most popular and powerful high-level full-stack frameworks for the python community, as they all include the basic Web features[79]. Plus, all of them, programmed and programmable in python, are open-source, have the powers of Web 2.0 version of the Web described above, follow the same architectural set-up (which will be discussed ahead) and equally suffice the stipulations set to suit the purpose of this research. However, picking at most one framework was compelling to represent the Web frameworks world by surveying its modeling methodology and support for spatial datasets. Due emphasis will be given to the database consistency, with the mind to lay a strong take-off ground to potentially address the Web-frameworks dealings in general.

Unlike any other Web framework, in web2py, it is possible to entirely develop, deploy and maintain Web applications via a Web browser and without the need to install it. This is the leading rationale which makes it preferred to other frameworks[74]. Based on the synopsis of[79, 95, 88] and evaluation made by the author, it is also found to have provably better and easy portability, speed, backward compatibility, security, web-based interfaces and architecture. As a result, we opt for web2py to particularly deal with it as a testing ground of the implementation on how to make functional adaptations of Web frameworks, following the redesigned architectural solution that this paper will proposed in Section 3.7 of Chapter 3. This adaptive implementation will be held in a more generic way to make it apply for all other frameworks with similar paradigm.

The salient advantage of Web frameworks is their architectural design. They follow the MVC structural pattern that enables the Web developers to clearly separate the three major concerns and basic components: the data representation piece (Model), the data presentation piece (View) and the application workflow piece (Controller) (see Figure 1.2)[74, 73, 96, 42]. The MVC pattern has proven to be a triumphant structure by Web developers[96]. The main advantage of this software architecture is its rapidity and easiness to make independent changes to one component of the application without introducing error and effect in the other two[42]. The model component, a sketchy imitation of the data model of the database technology, heeds the data structure and characteristically keeps track of all data traffic and interactions with the database back-end. The view at the front-tip generates an Hyper Text Markup Language (HTML) (and lately eXtensible Hyper Text Markup Language (XHTML)) formatted presentation to the user that mingles static and dynamic data derived from the Web and the database, respectively. At the middle of the architecture lies the controller segment in the form of functional algorithms that controls the execution and transformations of information flows between the two ends: the Model and the View.

1.2 Problem definition

Despite the advantage of using Web frameworks, their architecture follows an implementation-driven approach. The model configuration of the MVC (the M) lets Web developers directly begin server-side
scripting in an attempt to represent the reality. This set-up has a major limitation. It utterly ignores the bottom-up design-driven approach (conceptual-logical-physical design phases) of relational modeling (see left side of Figure 4); it follows a ‘non-database mode of representing reality to store it on the database’. This is to mean, it is not founded on a sound design basis since the standard ‘design-first’ principle for writing relational table schema is not explicitly treated. However, design-based modeling is considered to be an irreplaceably effective and efficient method for all applications involving databases[77].

Design is not a new concept in the Web database environment. Many literature sources overview it in relation to design of Web pages and design of the database they are coupled with. However, Web frameworks implement an impromptu data model, an approach termed as ‘quick-and-dirty’ in[48]. Web page contents are pushed to be stored in and pulled out of a database that has no pre-documented formal and structured design. “Although this procedure may appear pragmatic, such quick-and-dirty development method often results in massive quality problems and consequently in great problems in operation and maintenance”[48]. As an instance of such problems, one can pose a decisive question on how effective the consistency preservation approach is: the integrity component of the data model in databases.

For example, and particularly in web2py, an API named Data Abstraction Layer (DAL) is made to take care of an ad-hoc database realization under the hood, disregarding the design phases that might have enabled the DBMS to perform the task optimally (see right side of Figure 1.4). Such automatic configuration is a major downside of the Web framework engineering because it maintains only a small subset of spatial consistency rules which are the ones considered to be commonly used in many applications (see set A of Figure 1.5, \(A \subseteq U\)). There is no robust support for application-specific beyond-the-subset consistency rules, the majority of the rest of the consistency rules that are assumed to be uncommon such as the spatial and/or user-defined ones (be it spatial or not) (see set B of Figure 1.5, \(B = U \setminus A\), \((A \cap B) = \emptyset\), \((A \cup B) = U\)). If there is any means, it is merely an immediate solution that follows identical faulty procedure and repeats the same problem. While, led by design process, the DBMS could have handled the problem from its effectively. Monitoring and ensuring whether every user request to modify the data in the database obeys the constraint rules is an role that the DBMS can handle. The existing MVC architectural set-up of Web frameworks is not led by a well-designed model and ignores the rather effective alternative for consistency enforcement at the back-stage (see Figure 1.7).
1.2. Problem definition

While web2py communicates with the well-known relational DBMSs like PostgreSQL, mySQL, SQLite and so on, a set of validators inside the model which are triggered by actions inside the controller are made to manage the ad-hoc consistency rules on each database instance (see the highlights of Figure 1.6). Validators are built-in functional procedures stored as methods inside the framework package and actions are executable application functions created by the developer inside the controller, both written in the Python language. These pre-compiled methods handle only partial consistency rules, the so-called commons. This is insignificant compared to the range of consistency preservation possible by the matured DBMSs. The validators, that are included inside the model segment of MVC (the M) to play an integrity role, are not unequivocally defined to capture the application-specific requirements and semantics of the reality represented.
1.3 Research Objectives

1.3.1 General objective

This research aims at the re-engineering of MVC for a modeling solution and a re-development methodology for Web frameworks so as to make them design-driven and comprehensively manage beyond-the-subset spatial consistency rules, by providing a common coding language for both database and Web application design.

1.3.2 Specific objectives

The above general objective can possibly be split into the following sub objectives:

- To review the modeling and the consistency preservation approaches of Web frameworks and the databases
- To develop a theory of upholding spatial datasets within the domain of a Web application, using Web frameworks
- To build a re-engineered architecture that guarantees the proper integration of the consistency enforcement methods of the database technology into the Web frameworks, which has added support for beyond-the-subset spatial consistency rules
- To adapt the functional behavior of web2py, as a testing ground, in such a way that its consistency logics follow and implement the re-engineered architectural paradigm (by first enhancing it spatially)
- To compile an extensible framework library of the customized functions, together with their illustrative experimentation and demonstrations
1.4 Research questions

To achieve the above-mentioned objectives, the following specific questions should be efficiently answered:

- What are the commons and distinctions of consistency enforcement approaches in the databases and Web frameworks?
  - On what theoretical and/or mathematical foundations do they lie?
  - Why and where is the gap?
  - What needs to be inherited from each other? And
  - In which way and condition?

- How to utilize the Web frameworks for the development of spatial Web applications?
  - What is needed to be specifically considered for geospatial applications on the Web, when using Web frameworks? and
  - How to incorporate them within the realm of the Web frameworks?

- How is it possible to re-engineer the Web frameworks architecture so as to make it design-driven and support *beyond-the-subset* spatial consistency rules?
  - How is the spatial database design process incorporated as an integral part of the development process of Web applications backed by databases, when using Web frameworks?
  - How is the consistency preserving approach of the database technology shared by that of Web frameworks? and
  - How to make it handle *beyond-the-subset* spatial consistency rules?

- How are Web frameworks made to adapt themselves with the re-engineered architectural pattern?
  - What needs to be functionally modified in Web-frameworks to include the new engineering prototype?
  - What is the actual implementation of the newly formulated spatial structural solution, particularly with web2py?

- After the compilation of the functions based on the tailored architecture, how to make a practical test of all Web engineering achievements made?
  - Are all the defined problems rightly solved? If not, which ones and how to correct them? If yes, how are they exhibited?
  - What need further inquiry?

1.5 Innovation aimed at

The novelty of this study is a re-engineered Web frameworks’ architecture and an accordingly adapted software functionality in a manner that assures the design-driven spatial modeling practices are well-incorporated within it for a comprehensive consistency enforcement.

1.6 Related prior work

Much of the information about particular Web frameworks is available in no more than the form of manual and online documentation. These manuscripts don’t address the consistency issues independently, be it database-based or framework-based. For instance, neither the cookbook[74] nor the manual of web2py enterprise Web framework[73], two documentations prepared by Massimo Di Pierro, the creator of the web2py framework overtly deal with consistency rules. Also, the same applies to the Django, TurboGears, Grok and pylons documentations[42, 17, 18, 78]. They just document an implementation of the MVC and
attempt to only feed convenient information about their respective framework on how to fully understand the programming functionality and structuring.

The topic of design-based spatial consistency is well-addressed in the context of other database applications, like database-updating in Spatial Data Infrastructure (SDI) context. For instance, Milton Espinoza in [31] discussed the design and implementation of a Bolivian digital cadastral system to preserve topological and semantic consistency during updates. Sundaramoorthy also in [90] formalized constraint rules for spatial database updating and checking mechanisms of internal inconsistencies implemented on TOPI0 2D vector data by storing the constraint rules in a constraint repository. Similarly, Aynew in [7] tried to formalize the procedures for consistent updating in spatial databases with geometric, topological and thematic information by applying a topological model called cell-tuple structure. Although these works crucially add valuable inputs on approaches of design-based and consistent spatial data handling, none of them are Web-based.

A research question which is more affiliated with this project’s goal is raised by [54] on how to formulate a spatial data consistency protection mechanism that is applicable in spatial data editing on the Web. Moreover, many literature sources attempted to present a design-driven method to develop Web applications (spatial or non-spatial). For instance in [72], a framework that appropriately integrates a model of capturing the structure, behavior, navigation and presentation requirements of Web applications, and in [59] a framework for Web database design, were developed, demonstrating their successful test on real case-study. Plus, [54, 94, 12, 60] forward design approach within Requirements Engineering (RE) for a model-driven Web system development process. More related work, within MDA, the mapping of user-centered conceptually designed model onto the MVC logical model has been dealt by many studies such as [20, 65, 33]. (Here, the term ‘model’ is referred to the general representation of MVC itself rather than to the M component of the MVC). Whereas, the case of consistency in Web frameworks is not addressed in these research projects at all.

In [80], a Database-oriented Web framework (DOWF) approach is proposed as an outstanding methodology for creating flexible and powerful websites based on database systems, but in a view which is a bit different from the Web database system defined above. In addition, the literature also uses the terminology Web framework with a different meaning than the MVC-based framework definition applied in this paper.

Furthermore, this project is an academic work out meant for partial fulfillment of an M.Sc degree in Geoinformatics at University of Twente Faculty of ITC (International Institute for Geo-Information and Earth Observation), the Netherlands. The same project topic is shared by another fellow M.Sc student of the same department, Adam Kipkemei Menet, to investigate the consistency state of Web frameworks by pursuing a separate research approach, with the expectation to answer the same research questions upon the conclusion of the work.

1.7 Methodology adopted

As this project is dealing with a redevelopment of a software system, the core workflows of Unified Software development Process described in [47] was found an apposite method and is therefore referenced with some major adaptations made to it to fit the particular purpose of the project.

1. Survey of foundational concepts

- Review the theoretical and mathematical concepts with special attention to:
  - build a consistent geospatial applications on the Web using Web frameworks, with databases underneath
  - develop a methodology for the Web frameworks re-engineering so as to integrate design-based consistency manageability in spatial Web database applications

2. Spatialization of MVC

- Devise a generic outline that could serve as a guidance to spatially enhance Web frameworks
- Realize a spatial branch of web2py based on the outlined guideline, to test the designed spatializer guideline and make our dealings about spatial database consistency attainable
1.8. Organization of the thesis

3. **Design-driven** modeling
   - Carry out a RE within the spatial Web applications domain[39]
   - Define a use-case scenario that best exhibits the requirements and the Web database consistency functionality

4. Architectural **re-engineering**: 
   - Propose an architectural solution by redesigning the existing Web frameworks architecture in a way that the identified gap (regarding consistency handling) is properly filled and solved, and
   - Emphasize its applicability to database-supported Web applications in geospatial context

5. **Implementation**: 
   - Write and compile extended python functional codes that enables the web2py framework to implement the new re-engineered architecture
   - Note: This phase singles web2py out only as a testing ground of the redesigned architectural method, which will be framework-independently defined.

6. **Experimentation** and demonstration:
   - Examine the implementation, whether it enables the Web frameworks behavior follow the architectural solution and meet specified requirements
   - Once the validation verifies that the achievement is reasonably accepted, compile all deliverable write-ups and functional libraries prepared in the research process, take a corrective step otherwise
   - Prepare illustrative demonstrations, and save it together with the compilation above which will be the deliverable at the end of the project task

Literature Review and project write-up were considered to be the non-stopping, all-the-way through process until the final minute. Whenever there is a need to make a reference of the use-case, for a spatial data flow to and from the spatial database, a request to a Spatial DBMS would be made. The illustrative version of the methodology that will be followed by the paper is shown in Figure 1.8.

1.8 **Organization of the thesis**

This thesis is composed of eight chapters, including this introductory chapter, Chapter 1. They are organized in such a way that they follow the research method outlined in Section 1.7.

The chapter that proceeds, Chapter 2, is a review of concepts and recommendation for WDA modeling process. A stratification of major dimensions of Web applications were made followed by a recommendation of tool-based modeling mechanism. Plus, it introduces them into the Web frameworks way of practice. Chapter 3 takes the next step and puts a reviewed discussion on theories and approaches of spatial database consistency management. It highlights the theory of topological model that is considered to be a conceptual background of representations of geographic entities, including their constraints. A classification of spatial constraints in a way it is easy to specify and enforce them was also made in that chapter. Accordingly, a re-design of the client-server architecture being used by Web frameworks today proceeds.

Chapter 4 is a compilation of state-of-the-art principles usable to make a spatial enablement of the Web. It is also an implementation of the recommended one on web2py, following a clear-cut generic guideline we created. This chapter is considered as a stepping stone to make our non-spatial Web framework, web2py, ready to support a spatial dataset.

The second and important intention of this MSc research project was to make a check-up of consistency management methods on Web frameworks. The first step was, to carry-out a design-led modeling of a use-case at hand in Chapter 5. In Chapter 6, we documented out steps to implement the MVC re-engineering which we have figured out in Chapter 3. Thus, a Web application prototype were developed,
Figure 1.8: The method adopted, simplified
1.8. Organization of the thesis

mainly characterized by high data trafficking with the database, the main objective remaining the spatial
database consistency. Chapter 7 documents an illustration of the implemented prototype in a way the
enforcement of spatial consistency rules become visible. Finally, we closed the discussion by making a
summary and conclusion of our work, by forwarding some recommendations for further study and by
putting the known impact of our work on Web frameworks other than web2py.
Chapter 2

Review of Web database Application (WDA) Modeling

2.1 Introduction

The key intention of this chapter is to lay background notions on WE principles, especially founded on approaches that support tool-based modeling. For the sake of our discussion on manners of consistent management of the information content engrossed in a Web application, it is intelligent to start right from the separation of WDA concerns and dimensions, in a way to base them on models and model transformation principles. This is possible by means of MDA philosophy, the famous and highly recommended approach by Web and Computer Assisted Software Engineers. Hence, section 2.2 introduces the MDA, together with the details about three WDA dimensions.

Technology-aware development of the Web relies on the use of design tools in an attempt to model WDA concerns. As a widely preferred one among such tools, section 2.3 introduces the standards-based object-oriented modeling tool, UML. This supports the argumentation for accepting the UML-based Web Engineering (UWE) method in section 2.4. The last section 2.5 discusses on how to integrate the UWE, MDA and MVC triad. The chapter closes (Section 2.6) by describing where in the WDA dimensions the emphasis is to handle the spatial database consistency dealings.

2.2 Web-database application modeling process

Nowadays, the development of software tailored to build applications that could operate on the Web platform is supported by sound WE and Computer Assisted Software Engineering (CASE) principles. WE is an integrated discipline where Web software evolution and adaptation to new platforms and technologies is addressed by following software development principles based on models that are technology- and platform-independent[85], thereby enabling it to support the whole life cycle of the Web application development. On the other hand, CASE refers to the employment of tools for a semi-/automated generation of deployable computer codes that aid in the development of quality and maintainable software.

All engineering work (just like the Database technology) have a wealthy and long culture of employing models. They, in an effort to understand the complexity of real-world systems, base their underlying principles on models that simplify the entirety of the system: WE is not an exception. Models help to effectively reduce complexity by explicitly defining the basics of the system structure and characteristics to hide unwanted details, eliminate redundancies, separate key concerns and group interrelated items. In modeling, physical systems are abstracted by ignoring extraneous details while focusing on the relevant ones to help engineers’ reasoning capability in the development of the corresponding systems themselves[63].

This is to mean a design-based model that satisfies system- and user-requirements and at all does not depend on any platform of our choice should be first made. Such PIM should be separated from any platform-specific implementation details of applications, the PSM. The dependence on specific technologies should be delayed as long as possible[19].
2.2. Web-database application modeling process

The **Object Management Group (OMG)** (a specifications consortia to organize the practices of enterprise-scale software development and deployment, holding more than 800 member companies, organizations, and individuals) initiated the computationally cost-effective MDA proposal. MDA is a vendor-neutral approach to materialize the exercise of chaining successive models and the transformations between them, to generate code for controlled but flexible, efficient and automated software development in all phases of the process (from system specification through implementation and operation to maintenance)[63]. It is a model-to-model transformations whereby high-level abstract conceptual schema (CSs) of a PIM (which are also transformed from user and business requirements model, termed as the CIM) are transformed into a more detailed PSM. Then follows generation of deployable code which shall be directly executed by a specific platform (see Figure 2.1)[32].

![Figure 2.1: MDA conceptualized.](image)

In a nutshell, Web application modeling is glimpsed as three-dimensional artifact; it is diagrammed in Figure 2.2 and subsequently discussed. Nora and co. in [50] stabbing to reason about the stipulations of using separation of main concerns in modeling, they phrased out that “it offers advantages in the maintenance and re-engineering of a Web system as well as for the generation of Web systems for different contexts and platforms”.

![Figure 2.2: The Dimensions of Web Applications modeling (adopted from[87]).](image)
2.2.1 Dimension-1: The levels

In analyzing, designing and implementing WDAs, their continuously changing aspects are captured by first distinguishing the main concerns at different levels of abstraction, hence should be treated separately (see Figure 2.2). The storage and management of the data content engaged in dynamic Web applications is task of databases. All presentations, plus events triggered by the user are made via the interface to the application, which involves a two-way mapping between the information items in the application-side and the data items stored in the database[85]. For the sake of brevity, these concerns are discussed under three compartments below: content, navigation and presentation.

First level: Content

Content sets the representation of the information theme relevant to the application domain and their relationships (see Z-dimension of Figure 2.2). The source of such content can be eXtensible Mark-up Language (XML), databases and/or Web Services[34]. Data comes in different categories seen from different angles. For instance, it can fall into any of the following three architectural classes[29], while we are using the term in the sense of the first class.

- Content management system - Storage and management of structured data
- Documents management system - Storage and management of document-based resources, such as computer files
- Media management system - Storage and management of multimedia resources

A method that deals with problems related to Web-database application should primarily characterize the constituent information items by forwarding a data model that defines their relations, cardinality and constraints[85]. Content modeling in the way which is practical to traditional relational data modeling seems sufficient for static Web applications. Nonetheless, modeling only the structure of today’s multifaceted Web applications is judged partial and, in addition, modeling of the behavioral aspects is imperative[87]. (See the discussion about the third WDA dimension - Aspects in Section 2.2.3). Figure 2.2 highlights the piece in the WDA development process that particularly deals with spatial database.

Furthermore, it is commendable to re-mention here that the M is the piece in the MVC architecture that is vastly concerned with the ‘Content’ of the Web application. Though it is the screened-off-area to internally explore basic resources which are exploitable as consistency handlers, it is almost impossible to entirely separate the content sector of WDA from the other two levels described below.

Second level: Navigation

In [34] navigation is referred as “a functionality of actual application logic and encompasses all operations performed on the information space that are generally triggered by the user via the Web interfaces and whose execution is governed by some pre-documented rules”. It is concerned with:

- Hypertext – the proper linkage of the Web pages
- Transaction – the messaging between users and server-side applications, and,
- Dialogue/Conversation – the interaction of the application not only with human users but also with non-human ones, such as Web Services

This project may not deal conversational tasks because it calls for a design of further architectural set-up which is far from the target of our study, and will not be considered anymore. Many WE methods (including UWE, which is recommended by our study, see section 2.3) disregard it, however. Besides, the C is the piece in the MVC architecture that is interested with the streaming functionality of Web applications.
Third level: Presentation

Presentation is the general term that combines a gateway interface to place executable shortcuts to tools that trigger application functionality by users. Plus, it presents human-readable version of the data in an appropriate way that blends the artistic graphical design with Web engineering disciplines. Therefore its design method must deal with the \textit{abstract} design (application navigability) and \textit{concrete} design (layout, placement of interface elements and graphic appearance, such as color and font)\cite{85}. The V is the piece in the MVC architecture that is more or less interested with the interface-based presentation of the Web application.

2.2.2 Dimension-2: The phases

Which of the above three WDA levels should be modeled first depends on the project risks and specifications of the Web application. It is an independent decision to be made by the modeler him/herself. However, they all follow more or less standardized underlying steps. Every systematic software primarily engineered for the Web domain puts personalized and context-dependent requirements model at the start. Requirements model is a simplification of system functionality from the customers viewpoint\cite{52} in a way it satisfies their conceptual ‘requirements’.

It is in this manner that the Web system could easily be made adaptable to inevitable flexibilities that might arise after it has been already developed. After user- and application-specific requirements primitives and specifications are identified and analyzed in an early phase of the development process, the formal design-driven modeling of the information system pursues. It outputs a formal schema of the ‘Content’, which is a description of the whole process and is detailed enough to forward information to application developers (see ‘Phases’ fragment of 2.2).

In their doctrine about systematic development of Web applications, almost all WE methods recommend the importance of the sound and non-ambiguous conceptualization of the information system and its documentation into which they conventionally name is the conceptual schema (CS), first step in the design route. The design phase begins from the abstraction of the (platform- and technology-independent) implementation details by defining the problem domain and giving an overall view of the complex system from the users perspective\cite{19}, and is followed by the logical and physical ones, in that order. The logical schema (LS) is an interpretation of the CS on how to operate the abstracted system in a more standardized (but, again, platform- and technology-independent) language. And, finally, LS is translated onto a detailed dialect (called physical schema, PS) which is specifically adaptable and deployable onto a chosen application platform.

What is modeled should be made operational at the implementation step. Customization action proceeds, if necessary. This answers the question ‘how to adapt an application depending on the observed users privileges, roles and profiles?’, to build flexible application depended on user behavior and property. Note here that to make a smooth customization and maintenance to an implemented system, the adaptivity issue should also be handled and modeled separately. It is considered by some literature as a fourth cross-cutting dimension that has an impact on the other three dimensions. It takes us to the conclusion that whatever concern that precedes and follows the implementation stage calls for design-driven modeling.

2.2.3 Dimension-3: The aspects

WDAspect describes the characteristics of the Web system. It comes in two forms as Structure and Behavior (see ‘Aspects’ fragment of 2.2). All practices native to traditional Web database application development environment wholly focus on static \textit{structure} and its organization of the information at different levels of abstraction. However, credited to the modern agenda in the Web as an Engineering task, parallel to the structuring of the three levels goes an account of the \textit{behavioral} aspects such as business rules and user operations\cite{19}.

2.3 Modeling tool

UML\textsuperscript{TM} (latest version, UML 2.2) is a standards-based visual and textual object-relational modeling notation tool which is widely accepted as the \textit{lingua franca} in object-oriented software engineering world.
OMG\textsuperscript{TM}). Nearly all modeling methods for Web applications use it as their notation language, uncontested by any other tool\textsuperscript{[87]}. To build the static structural view of a software assemblage, the UML modeling mechanisms applied are class diagrams, deployment diagrams and component diagrams. And, the dynamic behavioral view, on the other hand, uses UML’s use-case diagrams, object diagrams, state diagrams, activity diagrams, sequence diagrams and communication (Interaction/collaboration) diagrams.

Figure 2.3: Hierarchy for metamodeling, adopted from\textsuperscript{[40]}. In line with the system development engineering task driven by models (and their inter-model mapping), the need to use standardized specifications to describe (or model) the models and modeling language themselves arises together. OMG\textsuperscript{TM}’s proposition in this regard is the Meta-Object Facility (MOF) as a ‘metamodeling’ architecture. “A metamodel is a precise definition of the elements of a modeling language, their relationships and the well-formedness rules needed for creating syntactically correct models”\textsuperscript{[50]}. MOF-based Modeling can be discussed at different abstraction layers (see Figure 2.3: starts from the bottom real-world objects (the data layer), via UML-aided models and meta-models (modeling UML itself), up-to the topmost meta-metamodel (MOF language itself), which are sometimes referred to as M0, M1, M2 and M3, respectively\textsuperscript{[40]}. A meta-model at a given layer describes the element at the next lower layer, i.e. M3, M2 and M1 are used to model M2, M1 and M0, respectively. Our usage of UML directs to M1, the first-level abstraction of real world objects by UML-led modeling tool 2.3.

Metamodels are good aids in the description of models, model transformations and automatic code generation in a CASE-supported design and model-driven Web system development. Mapping PIM model elements onto a target PSM model elements is made by marking each of the model elements in the source (PIM) models with an indication of their pertinent correspondingly mapped PSM model. This is specified by extending UML, which is not part of our dealing here (see Section 8.3.3).

Once the concerns of Web database applications are separated, what is to be addressed next is
the design process itself. Chapter 5 illustrates WDA design process held for testing redesigned Web frameworks’ architecture, following UWE method designed in Section 2.4. We employed Enterprise Architect (EA) UML tool for modeling. EA is a visually-aided modeling platform which is a commercial implementation of the standard UML by SparxSystems (http://www.sparxsystems.com/). To fully model the WDA, EA requires extension Metamodeling thoughts as discussed above. However, this is not the intention of the paper and only the modeling of M0-level objects by M1-level UML-led models has been taken care of (see Figure 2.3).

2.4 MDA-compliant WE method for WDA development process

There exist many WE methods that already have successfully integrated the MDA best practice in their development environment for the design-driven generation of powerful Web applications. To cite some, Object-oriented Web Solution (OOWS)[72], UML-based Web Engineering (UWE)[49][50], Object-oriented Hypermedia (OO-H)[9], Hypermedia Design Model (HDM)[35], Object-oriented Hypermedia Design Model (OOHDM)[86], Relationship Management Methodology (RMM)[43] and so on are among the Web Engineering methods that have structurally and practically suited the MDA-based development process.

Among all Web engineering tools that are MDA- and UML-compliant, this study uses UWE-based MDA-compliant Web Database Application development process. Why UWE? Because, unlike the other methods in the set, it is an approach based on interoperable open standards. “The main feature of UWE (OpenUWE) suite is its open architecture based on established standards that are supported by both open-source and commercial tools”[87]. It works with many fast spreading industry standards in addition to its compliance and adherence to the UML tool and MDA architecture, like OCL, XML Interchange Format (XMI) as interoperable and portable model exchange format, MOF for meta-modeling and Query-View-Transformation (QVT) as a model transformation language. Its UML-based visual and diagrammatic modeling technique aids systematic design and automatic generation that addresses the entire development lifespan of the Web systems development, starting with requirements models, through design models (including architecture and aspect models)[15].

Special views are defined for Web domain-specific elements using UML extension capability to broaden the UML metamodel, and they are named the UWE metamodel. Any CASE tool which supports the UML can be used to generate the UWE models[50]. UWE is expressed as:

- ‘Conservative’ or ‘lightweight’ extension of UML - because extensions are made lightly without modifying the modeling elements of the UML metamodel, by just adding additional features/associations to the existing modeling element of UML, i.e. the Class modeling element)[51], and;
- ‘Profileable’ - because UWE metamodel can possibly be mapped onto a UML profile[8], which is an advantage for the definition of the UML profile (specially, useful to easily import meta-models defined by other UML-compliant CASE tools).

Enterprise Architect provides a visual interface for ‘Use-case’, ‘Activity’ model and Class model specifications of RE phase and ‘Content’ model of the actual design phase of the WDA development process, as discussed in the following section. It also supports automatic content model transformations proposed by the UWE method. It is deficient UWE tool in the sense that it does not fully support modeling (and automatic transformation of such models) of the ‘presentation’ level of WDA design process, which is of less concern in our case. Hence EA still stays to be our preferred UML tool.

2.5 The design-driven WDA modeling process

Most MDA tools support the MVC design pattern of the Web frameworks[66]. Figure 2.4 is our design framework developed for UML-based Web Engineering, depicting the overall MDA-compliant WDA development process. It assembles chains of models and transformation between them, integrating the MVC triad. This design-driven WDA modeling process were used in Chapter 5 by picking a working use-case that deals with management of spatial data of bird species in Amazonia river.
Figure 2.4: UWE-based MDA-compliant WDA.
2.5. The design-driven WDA modeling process

2.5.1 The CIM: Requirements model

Firstly, using EA’s UML Use-case diagramming resource, the major agents and actors together with the descriptions of their interaction with the to-be built application are identified, from which the refined Web requirements (responsibilities and actions of the actors) are engineered by UML Activity diagrams later. Constraints should be explicitly specified as part of the use-case definition at this level.

For simplicity, the requirements should be addressed in two ways, users and functional. User requirements are analysis and reflections of the use and benefit expected to be gained from the developed system, otherwise they would remain fictitious and even useless. The functional requirements, on the other hand, algorithmically defines what functionality and use the Web would provide according to the identified user profiles. The proper selection of the functional units of the system is a great advantage, especially in the new architecture proposed in Section 3.7. Because, such functions are recommended for escalation of the Controller piece in Section 4.3.3 and for managing high-level application-dependent constraints in Section 3.6.1.

```
Figure 2.5: Use-case Relationships.
```

Features of use-case diagramming are plenty, but only three commonly applied relationships shared by use cases are treated below. The first one is the parent-child relationship between internal use cases called Generalization. In such a relationship, child use cases are known as enhancements from their parent use case but with their own specialized application significance. Generalization is revealed by a directed arrow with a white-fill triangle head pointing to the parent class (see Figure 2.5 for an example).

Extended to the UML Use-case artifact, UWE Use-case based requirements modeling also commonly employees two important stereotyped dependencies: «extend» and «include» (Again, see Figure 2.5). The former stereotype defines an extension relationship of the child use case adding on top of the current functionality and characteristics of the parent use case. Such relationship is shown by an arrow directed to the parent node with dotted pattern and «exclude» identifier. The latter stereotype depicts inheritance of the functionality of another use-case inside the model, i.e., the source use-case includes the functionality of the destination use-case as part of its business logic flow. It is also shown with the same arrow shaft as the extend relationship, but identified by «include».

Following the use-case drawing, one or more Activity diagrams should be prepared for each functional requirements identified (except, in our case, presentation requirements that entail from them). Each activity diagram is a simplified flow diagram of the functionality of the Web application launching from the ‘Application start’ point till ‘Application stop’ point. It includes clear demarcation of Web and database environments bridged by flow of transferable objects between them (see one of the Activity...
Chapter 2. Review of Web database Application (WDA) Modeling

diagrams in Chapter 5).

A set of Use-case and Activity diagrams define the initial CIM of an MDA approach that embraces the use-case characterization and the RE phases from Web Engineering point of view, and outputs a requirements model.

2.5.2 The PIM: Conceptual and Logical models

The PIM sets in motion from the conceptual design of the WDA with the help of UML class diagram, depended on the CIM Web requirements definition. The content, navigation and presentation should be modeled by means of their respective conceptualized representation. In a design-driven modeling, logical design phase follows the conceptual design. Therefore, all the conceptual models, wrapped by mapping rules, ought to be transformed onto their MVC-equivalent logical model, which is still a PIM-to-PIM mapping but more closer to the PSM compared to the conceptual model. Plain UML Class diagram is applicable to model the model component of MVC. Plus, one or more UML Sequence diagrams (SD) are applicable to model the controller component. Usually, for modeling the View piece of MVC, HTML-based web design editor tools might meet the basic requirement.

The following sub-section discusses the content modeling fragment of WDA modeling, the piece of the WDA modeling is engrossed in to model the data content that blatantly captures them jointly with their semantic constraints. For a better discussion on this matter, the reader is directed to Jan van Bennekom-Minnema’s work in [99], large-scale conceptualization of Land Administration Domain Model that treats constraints as part of the database schema. It also developed and assembled all-encompassing rules of model transformation, which we did not follow given the limited time for this MSc project. Plus, the Navigation, Interface and Adaptivity modeling are far beyond our discussion here, therefore are purposely excluded.

Content Modeling

A fresh, Content (data) modeling deals with the conceptual specifications of the domain-relevant information content of the to-be built Web application, resulting in Content (or Composition) model. Nevertheless, in UWE it very often is considered to include domain-entities required for customization of the Web environment (context model/profile) according to, and based on, the adaptations to the properties of users or user groups (user model/profile)[50]. Both Content models and Context models are visually modeled as UML Class diagrams. They should be treated separately, but their relationship should be meta-modeled using UML Association, which are derivations inferred from the domain objects defined in the details of the CIM requirements model (the UML Activity diagrams). However, the context model has less to do with database consistency issue and is another discourse that does not fit to the scope of this thesis project.

‘Unextended’ and unadorned UML Class element and semantics can be applied to model the content, since there are no any particular Web-domain specifics defined for them. Logical/structural aspect of the relations is taken as objects being instances of Classes, and their data fields (types) are abstracted as attributes. Whilst, the behavioral aspects are captured as operations, indexes, triggers, procedures etc. ‘Relationships’ is a UML way of specifying the semantic linkages that exist between instances of object classes basically as:

- Associations - Uni- or bi-directional connection between classifiers (instances of a class)
- Compositions - Whole/part relationship between model elements (classes) on the basis of some common grounds
- Aggregations - The so-called part class’s physical existence depends on whole
- Generalization - A model element being the general/specific class of another class, inheriting each other’s attributes and behaviors

Another important feature in UML Class diagrams is the cardinality (also called multiplicity) constraint between model elements. This is the mechanism to indicate the navigability between associated classes, expressed as the number of times each valid relationship is possible both or either directions.
Web Content modeling concern is much like the conventional database modeling, thus all practices of database design are inheritable to this step. Since our spatial database consistency dealings is wholly concerned with the design-driven data modeling (therefore, Content modeling), major focus was given to this partition of the WDA modeling (see the highlighted version of WDA modeling dimensions in Figure 2.2 and Figure 2.4. More on that, the M in the MVC architecture is the piece that handles the Content modeling and management task of the application. Hence, Content model maps directly to its equivalent model in MVC, the M.

2.5.3 The PSM: Physical model
This is the only stage where we talk in vocabularies which are decidedly restricted and specific to the chosen Web and database platforms. Web2py among the set of Web frameworks and PostgreSQL (made spatially-aware using PostGIS extension, see Section 3.8) among the relational DBMS are the preferred platforms to work with. Transformation of UML logical model onto physical one takes place to yield database constructs in the dialects of PostgreSQL, with some manual editions.

Here, the need for ORM contrast is two-fold. First, UML PIM objects, attributes and constraints specified in the modeling stage are mapped to their corresponding PSM relational representation as tables, attributes foreign key relationship and set of object-level, table and database constraints. Secondly, ORMapping between the two physical models (Web2py objects and PostgreSQL relations in our case here) should also be considered. Inversion of the database model onto Web frameworks model is as well required. This is to ensure the creation of a working M (of MVC) with its projected constraints handlers.

2.5.4 The M in MVC versus in MDA
To avoid unnecessary perplexity, it is intelligent to become aware of the deviation of the meaning of the term ‘model’ as it is introduced in MDA versus MVC. The usage of the term poses a slight difference with different contexts in action. Normally, a model (the M of MVC) is only a single compartment in the MVC triad that is solely concerned in the abstraction of the real-world reality to capture the information content and approximates the data model concept of databases, though it is not driven by design (see Section 1.1.4). While, a model stands for any design-driven abstraction of a system’s concern at a given abstraction level in MDA-based WDA development process.

For instance, three of the MVC compartments are abstracted by their corresponding physical MVC models in Figure 2.4 above. Also, let us have a glance at the hierarchical view of MVC shown in Figure 2.6. The model conception at the bottom level does not in principle accord with the model concept at the top level of the figure, all of them are abstractions perhaps.

2.6 Conclusion
In this chapter, an attempt to highlight the basics to hold the MVC design issue from the view point of Web Engineering was made. The major concerns and optimal possibilities for their interaction were separated in a manner that we can model them generically and transform them to a more specific detail, pursuing the MDA approach. Among other advantages, separation of the three ‘Content-Navigation-Presentation’ levels are helpful in two indispensable senses:

- One among the separated segments in the concern (the content) approximately accords to the culture of the matured database technology. Therefore, our discussion on best-practices of WDA design process from WE perspective enables us handle the spatial database consistency topic in an integrated and engineering way.

- More advantageous even is the more or less one-to-one correspondence mapping between the pieces in the WDA levels and MVC architecture (Content=Model, View=Presentation and Controller=Navigation). Modeling the WDA concerns is a suitable kick-off floor for mapping them to a more detailed MVC equivalent platform: like for example, ‘Content’ model onto ‘Model’ model.

One should also contemplate here that it is practically difficult, if not impossible, to wholly separate one WDA concern from the others. The need for their separation is only to ease complexities that entail
in course of action. Hence, keeping in mind that our objective is to assure every operation on WDA contents stay consistent, our due emphasis goes to the piece that handles the information content. All its conceptions are equivalent to the database, with its formal phases in designing and implementing it. The other two concerns (‘Navigation’ and ‘Presentation’) were dealt shallowly only because it is problematic to avoid them thoroughly and discuss about ‘Content’ singly.

Moreover, UML was the preferred tool in the modeling task leading to a UML-compliant Web Engineering method, the UWE. Lastly, by making an adaptation of the approach to an MVC and MDA approaches, a design for inter-model mapping are defined. In general, this chapter attempted to define the foundational concepts in WDA modeling. The actual WDA design process proceeds in chapter 5.1.
2.6. Conclusion
Chapter 3

Review of Spatial Database Consistency

3.1 Introduction

This is a review chapter of what we are going to assimilate into the Web frameworks engineering practices, the spatial database constraints management for an enhanced consistency. It begins by first giving a brief and practical definition of the term “consistency” within the realm of database technology in section 3.2. It is definitive to provide a formalized exposition of the principles in the wake of topological modeling theory (section 3.3), the key and simplified geometric representation of objects together with their relationships and constraints in space.

Dealing with constraints management calls for their across-the-board categorization in a way it is cushy to spell them out. Standing on the diverse taxonomic study forwarded by literature, the two approaches that differentiate constraints based on their spatial behavior and level of abstraction, in that order, are favored and discussed in section 3.4. The section that follows (section 3.6) is a summary of the state-of-the-art constraint modeling and enforcement approaches. The project uses a PostGIS extended PostgreSQL Relational DBMS to take care of the constraints outlined in the course of testing the Web frameworks for escalated consistency. Thus, its light introduction was made in section 3.8. Lastly, section 8.2 concludes the chapter by putting a squeezed version of it.

3.2 Consistency defined

DBMS is the powerful engine hidden at the back of many data-based applications. Among its overall management tasks, it maintains the correctness and accuracy of data content of the database by preventing possible sources of incorrectness and errors. For this, there are few mechanisms among which the following four are major control tasks[37]:

- Security control – prevention of an access and modification of data by unauthorized users;
- Concurrency control – prevention of inconsistencies introduced by concurrent access of multiple end-users;
- Reliability control – prevention of errors introduced by hardware or software failure;
- Consistency/Integrity control – prevention of semantic inconsistencies that mismatch with the explicitly defined integrity rules, usually named as constraints.

The fourth data accuracy prevention role of DBMS, consistency control, is the main concern of this study. Spatial database consistency (or integrity) is defined as the status of the database expressed by its self-describing characteristics and absence of logical self-contradictions. This way, it preserves the semantics of its frequently changing database content, by shifting it from and to consistent states.

In due course, consistency is a property of transaction in the sense that either successful execution of transactions or an otherwise prevention of invalid actions against the database is its main goal. Databases
utilize the general term ‘transaction’ to describe the CRUD operations and their executions by means of SQL Data Definition Language (DDL) and Data Manipulation Language (DML) statements. These operations are executed if and only if they accord to the consistency rules (constraints) defined as part of the database schema. The transaction is said to be valid if it has successfully been executed without any violation consistency rules.

If any of the actions of the operations of a transaction is found variant and invalid, the DBMS rolls the transaction back instantaneously or aborts it to make sure that the consistency of the database is preserved (or enforced). When a complex transaction is thought to involve very high computational cost, integrity checkpoint is marked at a point immediately next to where the last successful constraint has been enforced. The transaction goes back to this point for redoing. If successful, the database can be described as consistent and should unfailingly shift to another consistent state. A database that violates any constraint rules is not consistent, and is described as inconsistent.

A database consistency is as well indicated to be one of the four properties of transactions the databases technology possess, required for a reliable execution of transactions, acronymed as ACID (Atomicity-Consistency-Isolation-Durability). These properties are the powerful features that a DBMS must strive to assure the very existence and maintainability of the databases they manage. There is no sharp distinction among the ACIDs as it can be clearly seen from their following respective definitions:

- **Atomicity** - an “all or nothing” rule to ensure that either all or none of actions of the transaction is executed, each database transaction considered as being an ‘atomic’ unit of the database;
- **Consistency** - a rule that ensures whether the database stays consistent or not, altogether as discussed above;
- **Isolation** - a guarantee that there is no room for concurrent operations to access the data in a middle of a transaction state, for the sake of data consistency among transactions and performance of DBMS i.e Intermediate transactions have no semantics and the definition of database state holds only at two moments: before or after the successful execution of the transaction;
- **Durability** - a safety about the persistent and importunate storage of the successfully committed transactions against the database.

Again, [64] defined consistency as “Integrity = Validity + Completeness” to note that validity and completeness are the complementary elements of integrity. Validity here is applied to designate the exclusion of logically untrue information from the database and completeness on the other hand meaning the inclusion of all information that is true in reality.

### 3.3 Topological Model

In general, topological model is an essence that is entrenched within the the conception of spatial relationships such as distance between, length of and area of etc. Spatial management and querying itself is doable if topological relations of objects are explicitly defined and stored. Plus, spatial topological model is indispensable in the delineation of most constraints of spatial databases. Ahead of our discussion on constraints management itself in the context of spatial databases, it is recommended to discuss about the essence of topology and topological relations, hence will be covered here.

Topology is an area of study blended basically from geometry and algebraic set-theory that deals with the geometric properties and relationships of abstracted spatial objects that stay invariant as a result of continuous size/shape deformation, such as twisting or stretching. Its most familiar type is point-set topology, in which spatial relations are defined as pair-wise intersection between the boundary, interior and exterior of set of points. The formalization of point-set topological relationships is made by[26, 27], agreed upon to be the foundation of topology-based integrity control by literature.

Internally, a topology is made up of three basic elements; nodes, edges and faces (as seen in Figure 3.1) [44]. A database is said to be topologically consistent if and only if they are applied in the following standard sense.

1. **Node** - It is a single coordinate pair associated with the spatial location for that point, hence is represented by a point symbol. It might autonomously describe itself or bound edges or serve as
junction of edges (as it is defined below). Geometric representations of places like schools, national museums and clinics exemplify the former case and start/end points of a road network exemplify the later two cases.

2. **Edge** - It is a path between two nodes given as set of coordinate pairs of the starting node and the terminal node that bound it. Road segments and rivers are most commonly cited real world examples of edge topology. Sequence of the coordinates adds essential information into the linearity of features, the direction.

3. **Face** - It is a plane of closed and interconnected edges. It is expressed as a set of coordinate pairs of the bounding edges that defines its perimeter. Examples of real world entities that might be represented as faces include states, lakes and parcels.

Topological constraints which are not inherent within spatial databases require implicit or explicit specifications (see section 3.6 for an enhanced detail). The study of many of such spatial constraints are based on the concept of spatial relations as modeled by Egenhofer’s binary topological model, known as 9-Intersection Model (9-IM), an extended version of the 4-Intersection Model (4-IM) between the Interior and Boundary sets that includes the Exterior set.

Vector representation of an object in spatial databases falls in either of the geometric classes identified by OGC® in [70] (as seen in Figure 3.2). For simplicity, the building blocks in the hierarchy are considered by 9IM: point, line and polygon taken as 0-, 1- and 2-dimensional abstractions, respectively (Ø usually referred as having -1 dimension). The maximum dimensions of the possible intersections between the interior (denoted by ◦), exterior (denoted by −) and boundary (denoted by ∂) of two geometric objects (A and B) are braced in a (3 × 3) matrix, the Intersection Matrix. Firstly, the terminologies interior, exterior and boundary as defined in [26] follows (Figure 3.3):

- **Interior of X (X◦) –** “the union of all open sets that are contained in X, which corresponds to the largest open set contained in X, i.e. x belongs to the interior of X (x ∈ X◦) if and only if there exists an open set U such that x belongs to U, and U is contained in X (x ∈ U ∈ X).”

\[(x ∈ X◦) ⇔ (x ∈ U ∈ X)\]

- **Exterior X (X−) –** To define the exterior of a given point set, we first should describe its complement, the closure of X (X), which is “the intersection of all closed sets that contain X, which represents the smallest closed set containing X. An element x is in the closure of X if and only if every neighborhood of x intersects X, i.e., x belongs to the closure of X (x ∈ X) if and only if for every open set U containing x, the intersection between U and X is not empty (U ∩ X ≠ Ø).”

\[(x ∈ X) ⇔ (U ∩ X ≠ Ø)\]
3.3. Topological Model

Figure 3.2: OGC® Simple Feature Hierarchical Model
All points that are not within the closure fall into the exterior set. In other words, it is the set of point cells within the universe (Universal set, U) but that are outside (not elements of) the closure\[28].

\[(x \in X^{-}) \iff (x \not\in \overline{X})\]
\[(x \in X^{-}) \iff (x \in (U \setminus \overline{X}))\]

- **Boundary of X** ($\partial X$) — “the intersection between the exterior of X and the exterior its complement ($\partial X = \partial X \cap \overline{\overline{X}}$) and it is a closed set. An element x is in the boundary of X if and only if every neighborhood of x intersects both X and its complement.”

\[(x \in \partial X) \iff (\partial X \cap \overline{\overline{X}})\]

Furthermore, two additional rules of relationship hold between the interior, exterior and boundary of a given geometric object.

1. The intersection between the interior and the boundary of a point set is empty set;
\[(X^{o} \cap \partial X = \emptyset)\]

2. The union of the interior and the boundary of a point set is its closure set.
\[(X^{o} \cup \partial X = X^{-})\]

Assuming the interior, exterior and boundary of object A are given as $A^{o}$, $A^{-}$ and $\partial A$, and in the same way, the interior, exterior and boundary of B are given as $B^{o}$, $B^{-}$ and $\partial B$, respectively, the Intersection Matrix of Egenhofer’s 9-intersection model looks like:

\[
\begin{pmatrix}
A^{o} \cap B^{o} & A^{o} \cap \partial B & A^{o} \cap B^{-} \\
\partial A \cap B^{o} & \partial A \cap \partial B & \partial A \cap B^{-} \\
A^{-} \cap B^{o} & A^{-} \cap \partial B & A^{-} \cap B^{-}
\end{pmatrix}
\]
### 3.3. Topological Model

#### Figure 3.4: Simple topological relations of point feature with point, line and polygon features

<table>
<thead>
<tr>
<th>Relation</th>
<th>Expression</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Point-Point</strong></td>
<td>$\emptyset \neq \emptyset$</td>
<td>$\emptyset \neq \emptyset$</td>
</tr>
<tr>
<td><strong>Point-Line</strong></td>
<td>$\emptyset \neq \emptyset$</td>
<td>$\emptyset \neq \emptyset$</td>
</tr>
<tr>
<td><strong>Point-Polygon</strong></td>
<td>$\emptyset \neq \emptyset$</td>
<td>$\emptyset \neq \emptyset$</td>
</tr>
</tbody>
</table>

#### Figure 3.5: 8 simple Polygon-Polygon topological relations

<table>
<thead>
<tr>
<th>Relation</th>
<th>Expression</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Disjoint</strong></td>
<td>$\emptyset \neq \emptyset$</td>
<td>$\emptyset \neq \emptyset$</td>
</tr>
<tr>
<td><strong>Contains</strong></td>
<td>$\emptyset \neq \emptyset$</td>
<td>$\emptyset \neq \emptyset$</td>
</tr>
<tr>
<td><strong>Inside</strong></td>
<td>$\emptyset \neq \emptyset$</td>
<td>$\emptyset \neq \emptyset$</td>
</tr>
<tr>
<td><strong>Covered by</strong></td>
<td>$\emptyset \neq \emptyset$</td>
<td>$\emptyset \neq \emptyset$</td>
</tr>
<tr>
<td><strong>Covers</strong></td>
<td>$\emptyset \neq \emptyset$</td>
<td>$\emptyset \neq \emptyset$</td>
</tr>
</tbody>
</table>

[28]
Figure 3.6: 33 simple Line-Line topological relations[28]
The entries of the intersection matrix can be empty ($\emptyset$, -1) or non-empty ($\neg\emptyset$), and the non-empty ($\neg\emptyset$) ones can take any pertinent element of the set \{does not matter (* or -), point (0), line (1), polygon (2)\}. The maximum dimension is taken in case of more than one entry possibility. i.e. if there is a possibility of $\emptyset$ and point (1) to be the entries in the intersection matrix, then the point is picked. The following table 3.1 lists the possible entries into the Intersection Matrix in terms of dimension and the number of valid spatial relations as a result of the binary intersections between two geometric features (A and B), for each possible combination.

<table>
<thead>
<tr>
<th>Combination</th>
<th>$A \cap B$</th>
<th>Number of Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point-Point</td>
<td>$\emptyset(-1),\ Point(0)$</td>
<td>2</td>
</tr>
<tr>
<td>Point-Line</td>
<td>$\emptyset(-1),\ Point(0)$</td>
<td>3</td>
</tr>
<tr>
<td>Point-Polygon</td>
<td>$\emptyset(-1),\ Point(1)$</td>
<td>3</td>
</tr>
<tr>
<td>Line-Line</td>
<td>$\emptyset(-1),\ Point(0), Line(1)$</td>
<td>33</td>
</tr>
<tr>
<td>Line-Polygon</td>
<td>$\emptyset(-1),\ Point(0), Line(1)$</td>
<td>19</td>
</tr>
<tr>
<td>Polygon-Polygon</td>
<td>$\emptyset(-1),\ Point(0), Line(1), Polygon(2)$</td>
<td>8</td>
</tr>
</tbody>
</table>

The simplified Point-Line-Polygon abstraction can be grouped into six binary (pair-wise) combinations between two objects A and B in $R^2$ space as: Point-Point, Point-Line, Point-Polygon, Line-Line, Line-Polygon and Polygon-Polygon. For instance, considering the entries of the Intersection Matrix as being empty either ($\emptyset$) or non-empty ($\neg\emptyset$), there are $2^9 = 512$ possible simple polygon to polygon relations, out of which only 8 validly exist in $R^2$ space: disjoint, touch/meet, equal, inside, contains, covers, coveredBy and overlap (see Figure 3.5 below). The 9-IM for each relationship is directly below to its geometric sketch. Figure 3.6 depicts the 33 valid relationships between two line features, Figure 3.7 depicts the 19 valid relationships between a line and a polygon feature, Figure 3.4 depicts valid relationships of point, line and polygon features with a point feature.

Note: there are converse topological relations in the set, like “A covers B” means “B is covered by A”.

3.4 Constraints and their Taxonomy

Constraints are set of validity rules that are expected to maintain the consistency of the spatial database. Lots of attempts to provide constraints a formalized classification have been made, for instance by[89, 11, 30, 37, 56, 81]. Though not so common, some literature sources segregate constraints as spatial and non-spatial. By non-spatial they meant the alpha-numeric constraints that were native to traditional databases, existent ever before the advent of spatial applications. The first DBMSs were non-spatial, because there was no any spatial composition in the data they manage and constraints associated with them.

The first discussion on the categorization and implementation of constraints followed by this paper is based on the most common Cockcroft’s simplified taxonomic study of constraints in [30], which is based on the concept of improving data quality by enforcing constraints against the database on data insertion by the user. It considers the spatial behavior of objects as a yardstick, and it lists topological, semantic and user-defined constraints.

Next, grounding on levels of abstraction, we also come with another class of constraints thought to be easier for our constraint specification: object-level, table-level and database-level. Figure 3.8 frames the detail, and the discussion follows.

3.4.1 Spatial constraints: based on spatial behavior

As their name signify, spatial constraints describe the location related behavior of the entity represented, spatial data and their attributes. Storing all the geometric relations in the spatial databases is computationally expensive, therefore they should be verified for a better consistency of the database[90]. Topological, semantic and user-defined constraints are the three classes of spatial constraints.
Chapter 3. Review of Spatial Database Consistency

Figure 3.7: 19 simple Line-Polygon topological relations[28]

Figure 3.8: Constraints categorization
3.4. Constraints and their Taxonomy

**Topological constraints**

Topological constraints are the general rules of topology [3.3] which are inherent to any topology-represented spatial object, with their possibility to be verified at an early stages of data modeling. Basically, they are there to maintain the topologically correct shape of geographic features by definition. Effort towards highly standardized and functional specifications for inherent topologically constrained spatial features has been made by OpenGIS® Consortium under Geometry Object Model in [70], based on the the essence of topology as introduced in [44] (see Section 3.3).

**Semantic constraints**

Cambridge learner’s dictionary [45] defines semantics as the ‘study of meanings’ and in spatial databases it is concerned with the true meaning and connotation of the abstraction we give to spatial objects. Semantic integrity constraints restrict the validity of objects in relation to their true properties, by rejecting data that contradict the semantic definition given to it. Such definition is expressed in terms of topologically valid relations that should prevent erroneous geographic relationships of features, for a well-functioning consistent spatial database.

Simple examples in this case are: two linear representations of extended road features that do not meet, line representation of parcel lands, inter-country boundaries that overlap, a building in the middle of a railway line, road crossing a lake etc. are semantically meaningless, for instance. Unlike topological constraints, which are inherent, detection of semantic constraints depends on how well they are specified.

**User-defined constraints**

Relational Database Management System (RDBMS)s give users the opportunity to define their own business rules (known as user-defined rules) for a better consistency of the database dependent to their own context and perspective. The definition of specific rules (such as for a legal purpose) which are subjectively dependent on who is defining them are left to the user’s for specification. “many rules that exist in the real world do not have their counterparts in the data models and have to be specified explicitly[93]”. They are representations of domain-specific knowledge [104]. These can be about restrictions about what should or should not exist between geometric objects. How far from noisy sites should a hospital be constructed or how wide should a road be, can be expressed in terms of topological relations and depend on how users understand it.

3.4.2 Spatial constraints: based on level of abstraction

Most commonly, consistent database management requires constraint specification and validation at different levels. They are discussed as follows:

**Object-level constraints**

Object-level constraints are applicable to a lower level attributes or columns only. Such constraints cannot be stretched beyond the mere frontier of the given object. We can enumerate the following example classes of constraints applicable at object-level attached with some exceptional cases.

- **Domains:**
  
  Domain Constraints check whether the attribute values are drawn from range of values, types and structure (which are enumerations of the only possible attribute entries) specified in a certain domain. Simple examples are: Area field should not be negative (Area > 0), Geometry should be of POLYGON type only (POLYGON(x)), Age should not be negative and should not exceed 100 (0 < Age ≤ 100) etc. These can be expressed in First Order Logic (FOL) as:

  \[
  (\forall x \in \text{Area})(x > 0)\\
  (\forall x \in \text{Geometry})(\text{POLYGON}(x))\\
  (\forall x \in \text{Age})(0 < x \leq 100)
  \]
Chapter 3. Review of Spatial Database Consistency

The domain specification can also be made by creating a separate relation for it. Like, assuming p is an attribute of relation $R_1$ (denoted by $R_1.p$) and q is an attribute of relation $R_2$ (denoted by $R_2.q$, list of range of possible values for p), then:

$$(\forall x \in R_1.p)(x \in R_2.q)$$

- **Entity integrity rules:**

  The notion of these rules, what are also called PRIMARY KEY and UNIQUENESS, is to give unique identification to each instance of an entity type, and it should not be null. In a given relation ‘owner’ which holds information about owners of parcels, if the owners are identified by the identification number (id attribute, owner.id), there should not at all exist two instances of the same id in the same relation. This means in FOL notation:

  $$(\forall x, y \in \text{owner})(x.id = y.id)(x = y)$$

  A relation may have a single unique identifier column which serves as a PRIMARY KEY, in that case we are referring to constraint that applies to a given attribute. But, there are times when just a single column does not suffice as a one-per-tuple identifier and combinations of two or more columns should be made to recognize each instance (known as COMPOSITE PRIMARY KEY), which makes it go beyond a single attribute, and are considered at the next level, Table-level.

  Also, the concept of UNIQUENESS can be seen globally (not necessarily within the object domain) or locally (only within the given domain). The later case goes beyond a mere sphere of influence of an object and is rather considered at the level of table. For the sake of local UNIQUENESS, a tuple (record) should be checked whether or not it is unique within a classified group of objects. Like for instance, two records about taxonomic order of species of the same genus should not assume identical values, but does not matter if species of other genus class consider that value. Global UNIQUENESS, on the other hand, is a check on the uniqueness of the assumed value within or outside the specified class. To use the same example, one may say scientific name that identify a species should not be shared by any other species.

- **Attribute structural constraints:**

  Attribute structural constraints declare whether null-value, single-value and multi-value is allowed for an attribute or not. Considering the parcel example above, the not-null structural constraints (also called, Mandatory Attributes) can be expressed as:

  $$(\forall x \in \text{owner.id})(\neg(x \text{ is null}))$$

**Table-level constraints**

Table-level constraints are one-stairs higher than object-level constraints, seeing that they make a fuss of at least two objects of a given relation. A common example includes:

- **Composite PRIMARY KEY and Global UNIQUENESS:** This implies to Entity integrity rules when more than two columns are made to uniquely identify entries of a given table and when uniqueness goes global, as discussed under the second item of the above object-level constraints, 3.4.2.

**Database-level constraints**

At an even more advanced level, we find constraints that involve two or more tables within the database. The most common example at this level is:

- **Referential integrity constraints:**

  This is about the establishment of references to relations by means of a SECONDARY/FOREIGN KEY for creating semantic relationships between and among them. This is the central concept of relational data model. REFERENTIAL INTEGRITY constraints state that unmatched reference FOREIGN KEY are not allowed i.e. Each FOREIGN KEY reference should contain a uniquely identified entity (usually its PRIMARY KEY) in the referenced master relation. Unlike PRIMARY KEYs, SECONDARY KEYs can partly assume null-valued attributes. Relational DBMSs preserve the referential integrity by trying to cascade all modifications made to the master relation into
3.5 Constraint specification

As rules definable to guarantee the integrity (consistency) of the spatial database, if constraints are not explicitly specified and documented, how could they be enforced? In Relational DBMSs, constraint specification is considered to be an integral part of the definition of the database schema, an output of a design process. Database schema is a set of relation schemas which in turn are sets of relation names, their list of attributes and set of constraint rules that must be satisfied in any case by a valid instance of a relation. The constraint rules are needed to capture the real-world ‘semantics’ and create a well-functioning representation of the reality. The rules are defined in such a way that they guarantee the consistency of the database. If constraints are not specified for validation, the integrity of the data will be negatively affected and jeopardized. Their specification comes in either of the following two forms[90]:

- As Built-in constraints – packaged inside the DBMS as rules that can be, when needed, retrieved and activated by the user (For instance, the stored procedures of Geometry Engine, Open Source (GEOS) module inside PostGIS)
- As User-defined constraints – rules and triggers explicitly defined for particular database transactions or methods at an implementation stage.

Our intention here is on the user-defined application-specific constraints, their specification should come as part of the database definition. Our preferred way of specifying constraints is by following design-compliant approach of segregating constraints discussed in Section 3.4.2, Constraints at three levels of abstraction.

3.6 Constraints enforcement

The approaches to model and verify spatial constraints are very diverse. However, technically speaking, constraints have three options of implementation in terms of where in the application development process they could be applied: at Data model level, at DDL level and at application level, called inherent constraints, implicit constraints and explicit constraints, respectively. Figure 3.9 depicts each levels where constraint implementation possibly applies. Their definition as alluded in[30] follows:

- Inherent constraints - are intrinsic part of the defined object classes in the data model itself, hence they are not directly specified for enforcement in the database schema because they are indirectly applied by defining the model constructs. Rather good example in this regard are the constraints
inherent in the open geographic standards: OGC® Simple Features Specification for SQL[70], Geographic Markup Language (GML)[71], International Standards Organization (ISO) Spatial Schema (ISO 19107)[44] and so on.

For instance, OGC® Simple Features Specification[70] gives functional definitions to geometry, topology and spatial operations. As an example, OGC’s definition for valid Polygon includes, among others; it should be coplanar surface and topologically closed, there should be no cut lines (spikes or punctures), its interior is connected point set and so on. By defining a feature class as polygon, one is specifying these constraints inherently.

- Implicit constraints - DDL are used to specify the constraints which are transformed from the constraints defined conceptually in the design process, using an object-orienting modeling tool. For instance, [10] argues about the importance of an earlier specification of integrity constraints in spatial databases using object-oriented data modeling tool (recommending OMT-G, Object-modeling Technique for geographic applications) that defines the relationship between the spatial information, spatial relationships and integrity constraints. Similarly, [41] developed a Geographic Relational Data Model that integrates topological constraints.

- Explicit constraints - Locally specified by the application programs that use the database, in a procedural (writing statements that encapsulate triggers for checking the consistency of each update instances) or declarative approach (storing constraints in a suitably encoded form).

Moreover, different methods of constraints handling for different classes of constraints are forwarded by[11]. It suggests the usage of active repository (such as Oracle CDM Ruleframe) for user-defined spatial constraints, object-oriented approach for semantic spatial constraints and integration of logic programming and databases for topological spatial constraints enforcements. The same literature discusses about the use of a constraint repository for a controlled error warnings and data quality metadata reporting and Sundaramoorthy[90] applied this approach for spatial database updating and internal inconsistencies checking implemented on TOP10 2D. This mechanism uses a user interface and automatically translates it onto DDL statements as part of the database model or as queries for a Geographic Information System (GIS) system[104].

[89] developed an integrated interface-based framework to improve the consistency of spatial databases of (vector) data giving its own categorization of the spatial constraints as structural, geometric and topo-semantic errors (constraints). Another framework on how to use formal ontological definitions of spatial constraints using SWRL (Semantic Web Rule Language), a marriage between OWL (Web Ontology Language) and RuleML (Rule Markup Language), has been devised by[61]. Also, a proposition to use CDT (Constraint Decision Tables) that uses ECA (Event-Condition-Action) rule for constraints management is devised by[102]. The reader is referred to the respective documents for more on these subjects.

Most importantly, UML-led MDA-compliant way of vivid declaration of spatial behaviors at the data modeling level is possible to be materialized using the formal human- and machine-readable OCL subset of UML. OCL (latest version, OCL 2.0) is a standard, platform-independent, close-to-natural and descriptive (notational) language to express constraints of objects in a given model, at the level of classes, attributes, associations, and operations(OMG™).

OCL has a generic support for many non-spatial constraints. Geographic extensions are proposed for the standard OCL grammar by making it support an added type GEOMETRY and its operations, [61] (the GeoOCL), [1] (‘Spatial OCL’ for modeling Wireless Ad Hoc Networks), [75] (‘Spatial OCL’ for Environmental Information System (EIS)) and [76] (GIS extensions to OCL for Agricultural Information Systems (AIS)). Additionally, [22] attempted to integrate Egenhofer’s eight region-to-region topological 9IM relationships into OCL operations. MDA prototypes for Land Administration Domain Model were developed by [99] and [2] that map PIM constraints as part of the entire domain model (source, Object-oriented) onto their appropriate database PSM (target, relational PostgreSQL/PostGIS database). All OCL enhancements are propositions and are not yet implemented by UML-compliant modeling tool, including Enterprise Architect. Therefore, time-wise, manual constraints specification has been considered economic and applied for our forthcoming use-case in Chapter 5, as part of the adopted.
3.6. Constraints enforcement

3.6.1 Preferred constraint implementation approaches

More precisely, constraints can partially be specified directly in the data model and incorporated within the database schema to automatically be enforced during database creation. Those constraints that could not be enforced at database creation stage are enforced later at an implementation stage by constraint functions, checks, methods and triggers at run-time during data insertion, deletion and update. Find a detail about this below.

1. Checks:
The simplest way of implementing constraints in RDBMSs is by defining check constraints that must apply to each entry row in a table. These checks return Boolean as TRUE, FALSE or UNKNOWN depending on the condition set. The most common example is the NOT NULL constraint that prevents null values from populating the table. Here is the line that adds this example check into, let us say, a ‘name’ column of a ‘species’ table in PostgreSQL.

Basic syntax:
ALTER TABLE table_name ADD CONSTRAINT constraint_name NOT NULL (column_name);

E.g.:
ALTER TABLE species ADD CONSTRAINT name_not_null NOT NULL (name);

2. Triggers:
Again, RDBMSs most commonly use database triggers as one way of validating constraints. They define a method for conditional enforcement of constraints as, for example, implemented by [56, 23]. This means, a procedural code is embedded with some constraints as pre- and post-conditions that should be fired before/after events of data insertion, update or deletion happen. Triggers can be made to alertly track the transaction and react automatically when illegal actions (like insertion of invalid data or unauthorized deletion of data) are detected by the system. The expected reaction is binary, either True in case the condition set has been met or False otherwise. Hence, Triggers return values of Boolean type.

For instance, in PostgreSQL, languages like PL/Pgsql, PL/Perl, PL/TCP, PL/Python etc. are few of the languages usable to define such application-specific functions on the server-side. If we take the case of PL/Pgsql, the definition of triggers involves two important steps. Firstly, the main body of the function that returns TRIGGER should be defined. Secondly, an assignment of the defined trigger function to a table, given some pre- and post-conditions, takes place. See the example syntax below, which is very simplified way. Find the details in [38], the online documentation of PostgreSQL.

(1)
CREATE OR REPLACE FUNCTION add_family() RETURNS trigger AS
BEGIN
  [main body of the database function]
END;
LANGUAGE plpgsql;

(2)
CREATE TRIGGER add_family BEFORE INSERT OR UPDATE ON family
  FOR EACH ROW EXECUTE PROCEDURE add_family();

The first types of triggers are only one-stair higher that check constraints, and are termed ‘Check triggers’. Check constraints that involve more that two tables could not be verified by checking
method described above. Here emerges the need for Check triggers. One may define a database-level rule that constrains species distribution geometries to fall fully within a legal bounding box, for example. This is one of the bold example constraints discussed in Chapter 5. On the other hand, the second types of triggers, ‘Trigger functions’, are database functions more advanced than mere checks, but still return Boolean value.

3. Callable functions:

At an even higher level, user-specific database-side procedural functions can be made to extend database functional units using proprietary languages, then stored and called later from an application or transaction. Their definition grammar is similar as in the first step of defining triggers, but the return value necessarily should not be trigger (can be TEXT, INTEGER, ARRAY or whatever, refer manual of PostgreSQL, [38]).

Such functions are termed as ‘callable functions’ here to indicate that we want the Web application to call them later directly from the controller. This entails from the intention we have to keep the database as self-sufficient as possible. Chapter 6 illustrates the situation.

The other scaled benefit of these functions is that there is no need of defining an independent function of this kind. They can be integrated and nested inside server-side database function that are foundations of the advanced Controller actions as pointed out in Section 4.3.3, recommending this approach,. The same server-side function can be made to handle both situations, define application-specific controller actions and complex functions. Complex constraints that could not be handled by simple checks and triggers as defined above can be embraced inside database functions used by the Web application.

3.6.2 Constraints violation and response action

Plainly, enforcing constraints means either it is in the form of violation prevention or violation detection[37]. While an instance of constraints violation encounters in the middle of a transaction, the constraint is enforced before the transaction commits and the error propagates. It is prematurely aborted (or rolled back to top-level checkpoint), then we say violation is prevented. The database is still in the same state as it was before the commencement of the transaction, therefore, it does not need any recovery. Constraints enforcement after an update has already been undergone is termed as violation detection. The transaction needs to be undone after its abortion.

The safest way is however to prevent erroneous data from entering onto the database than to take corrective measures once they contaminate the existing data. By this being said, literature more often discuss two common classes of system responsive actions to constraints violation.

- Error message - issuing an instant pop-up error message, usually asking the user for further action;
- Transaction abort - The safest way is perhaps to abort the transaction and take necessary back-up action, if any.

3.7 Architectural re-engineering

Grounding on the assessment of the above state-of-the-art consistency-preserving mechanisms, the Web frameworks architecture shown in Figure 1.3 has been redesigned and adopted to the recommended architecture shown in Figure 3.10. The reason is to put constraint specification and constraint enforcement approaches together with the Web application development procedure. So that, the proper marriage between the practices of Web frameworks and databases become materialized. It is this re-engineered architecture that our prototype creation pursued in Chapter 6. In essence, it is a cloning of concepts summarized in 3.

What is new here is the additional room it provides for the specification of constraints at different levels of abstraction (Object-, Table and Database-levels, discussed in Section 3.4) and their enforcement at different degree of constraint complexity (check constraints, Triggers and callable functions, in ascending order, see Sub-section 3.6.1) above. Some constraints might also be enforced by handlers that have already been implemented in the existing structure of the Web frameworks. They were categorized as Legacy constraints and included just below the check constraints in the hierarchy of the new architecture.
3.8 Introducing the PostgreSQL Relational DBMS (extended with PostGIS)

An example of legacies includes the UNIQUENESS and NOT NULL constraints (see Section 6.2.1 for a better detail).

![Diagram](Image)

Figure 3.10: Proposed marriage between design-driven modeling and MVC architecture

3.8 Introducing the PostgreSQL Relational DBMS (extended with PostGIS)

PostgreSQL [38] is an advanced and OpenSource object-relational DBMS created by professor Michael Stonebraker at University of California at Berkeley (UCB) between 1986-1994, as a follow-up project of its predecessor (INGRES). It is based on classical theories and concepts of of Relational DBMS, with panoramic mission of integrating ‘object-relational’ technologies. For a better discussion, refer the ultimate documentation of PostgreSQL[38].

PostGIS [58] is an addition of spatial branch to PostgreSQL by introducing new data types, operators and functions to it that makes it accommodate and manipulate geographic objects. In compliance to the geometric data types and functions of OpenGIS® Simple Features Specification For SQL[70], it spatially enables the PostgreSQL database server. Currently, it contains about 700 full-fledged GEOS functions that allow spatial query, analysis and integrity operations, such as topological containment, overlap, intersection and so on. The PostGIS-extended PostgreSQL database server consists of a rich resource of stored procedures that support the execution of all the basic constraint rules categorized under constraint taxonomy section 3.4 above.
3.9 Conclusion

This was a chapter where we summarized topics related to spatial database consistency. Intending to review established best practices, the first thing to deal with was to supply a functional definition of the term ‘consistency’ in the context of database-enhanced applications. For that, Section 3.2 tried to locate consistency among the most commanding control actions and properties of transactions that DBMSs carry out. For better understandability of rules of consistency, which are called constraints, the underlying essence of topological model were discussed in Section 3.3, accompanies by a discussion about topological relationships within and between geometric objects. Hence, node, edge and face were found to be the building block all topology-based representation of geometries.

While surveying different research studies, we encountered with miscellaneous ways of constraints segregation. In Section 3.5, preference was made to pick the only minimalistic two of these diverse classification schemes. First, we looked at the one that bases itself on spatial behavior of entities; as topological, semantic and user-defined. And, the second depends on increasing level of abstraction; object-, table- and database-levels. These were found to be useful for simplified design process. There is also a diverse culture of enforcing spatial database constraints. We attempted to make a summary of these useful practices in Section 3.6. However, to enforce the constraints specified at the three levels of abstraction, we proposed to segregate them into different classes based on their increasing degree of complexity, going from simple to complex, in Section 3.6.1. Our design process in Chapter 5 uses this approach to manually specify the possible constraints involved in an operational use-case. With this, we make a re-arrangement of the Server-Client Architecture most Web application development tools follow, including Web frameworks, to integrate our constraints specification and enforcement methods into the whole set-up for an advanced consistency control, in Section 3.7. Later Chapters trailed this newly proposed approach.
3.9. Conclusion
Chapter 4

Spatialization of the Web Frameworks

4.1 Introduction

Non-standalone applications within the spatial sphere such as GIS, Location-based Services (LBS), Remote Sensing (RS) and the like greatly require the Web for easy accessibility and transferability of their content. The geographic data and GIS soft/hardware tools are seamlessly flocking to the Web in a format accessible and serviceable by all available computing platforms. They require complex tools and add-on environment than ordinary applications. In the last few years, spatial Web Services gained massive attention and recognition paralleled by high-speed scientific advancement both in terms of cost and efforts. This permits interactive space-related applications occupy the Web environment transparently, for Web-centered ‘GIS’ing and geographic data conveyance services.

Despite the openness and availability of diverse range of options to develop spatially organized Web database applications, the spatial culture around Web frameworks community is pondered as being very meager. The term ‘spatial’ or ‘geo’ is less-common in the Web frameworks world. GeoDjango, a geographic extension to django Web framework, is a pioneer in this regard, however.

This chapter instigated the key notions to answer the ‘how-to?’ question of spatializing Web frameworks. Primarily, we put an account of the standard concepts of ‘Spatial Web Services’ followed by a section that specifies the discussion to ‘the Spatialization of Web frameworks’, in Section 4.2. By grounding it on a meta-survey of the architectural design of the GeoDjango, a generic guideline that adds spatial dimension to each components of MVC was devised and implemented on web2py Web framework in Section 4.3, to set it with the nitty-gritty for the ‘CONSISTENT WFS-T application’ developed in Chapter 6.

4.2 The spatial Web

Spatial and non-spatial data are not totally detachable and are considered two sides of the same coin. The spatial artifact of services and applications related to digital resources assembles a previously nonexistent but valuable dimension. What makes a set of data spatial? It is the fragment that fanfares its location-based behavior which noticeably is central to every human activity. Such arrangement may adopt the industry-based Web customs and tools directly but they should be escalated by spatial annexes that activate the added spatial dimension.

Before assembling the pieces of the puzzle, it is fundamental to become aware of the time-honored fact that there are three basic dimensions to a spatial data in a broad-spectrum context:

- **Spatial** (locational) dimension - answers the **where?** question by modeling characteristics pertaining to locational aspects of objects in space, as a field-based raster or point-set vector format

- **Thematic** (attribute) dimension - answers the **what?** question by giving the data a definitive attributes pertaining to its behavior (much like the long-established data attributes)
4.2. The spatial Web

- **Temporal** (time) dimension - answers the **when?** question, displaying the time-dependent behavior of the data.

The first two facets apply to most types of spatial data, and are almost totally inseparable. The last dimension is supplementary to the two by sequentially capturing, storing and displaying spatial information, time being the decisive variable. The temporal characteristic of the spatial data requires meticulous treatment and is beyond our intention in this study.

### 4.2.1 The SOA triad

The Web is overcrowded by an extremely diverse practices that go behind the specificity of systems and developments over a distributed computing environments of data formats, vendors, platforms, operating systems, supported databases, coding languages and so on. It, as a universal and common platform for data sharing, offers technical and semantic interoperability and transferability by means of ‘Web Services Architecture’ that is designed to loosely accommodate such exceedingly diverse and specific Web culture. This loosely-integrated architectural suite as a rule is named the SOA. It modularized Web applications as services. By so doing, it greatly increased semantic access and processing capability of spatial content [62] by its diverse users. Programmers of SOA-based consumable services extensively use a common and transferable standard language (XML with GML being its geographic extension). Web Services Description Language (WSDL) speaks for itself, it is a language that describes Web services bridged by a standard protocol called SOA communication Protocol (SOAP).

Technically speaking, Web accessible and exchangeable spatially-aware applications regardless of location, platform, operating system, or language are known as ‘Spatial Web Services’. Industry-based Web services rely on SOA arrangement, involving three active agents: Provider, Requester and Broker (Figure 4.1). The underlying chain of **Publish-Find-Bind** principle guides the service operation, in each operation involves client-server architectural design (as discussed in section 1.1.3). Firstly, the server (service provider) announces (i.e. publish) its metadata detail about its service capabilities on the catalog and registries of the middleware (i.e. broker) Web portal with semantically enabled search engine, from where the service requestor/client queries (i.e. find) the kind of available service it demands. And finally, the requestor consumes (bind a contract) the service of its interest. Individual SOAs tie one another using service orchestration/chaining principles.

![Figure 4.1: SOA involving Publish-Find-Bind operations](image-url)
4.2.2 OWS: OGC® Web Services

OGC® Web Services are self-contained and standard versions of the Web services with spatial enablement of HTTP based SOAP and WSDL interfaces to access spatial data from distributed spatial databases via a thin-client (usually, a Web browser). These industry-level spatial Web services support the underlying principles of interoperability and SOA-compliant service orchestration. Figure 4.2 exhibits the ultimate relationship of SOAP-founded services within a OWS server-client architecture. To put some lights on it, at the bottom level appears storage component in the form of database or file. A Web client (for example, which is spatially enabled by JavaScript maps) gets access of the spatial servers by means of the data types supported by services provided. In each line of access, external Web servers (e.g., Google Earth API) might be used to enhance the service provision. This arrangement is entirely supported by the client-server architecture (see Section 1.1.3). A baseline OWS example includes:

- **Web Map Services (WMS)** – displaying geo-registered multi-layered interactive map images (in different formats: Joint Photographic Experts Group (JPEG), Portable Network Graphics (PNG) etc.)[103, 16]. WMS server listens the following main standard requests:
  - **GetCapabilities** – returns XML parameters and description about the available WMS and the available layers
  - **GetMap** - returns a map image of a specified format and spatial extent
  - **GetFeatureInfo** – retrieves feature-level detail information about a specified geographic extent of a map
  - **DescribeLayer** – retrieves a description of a single map layer
  - **GetLegendGraphics** – retrieves graphics representation of a map layer displayable at map legend

- **Web Feature Service (WFS)** - querying or executing some operations on one or more features[100], in GML, Well Known Text (WKT), Keyhole Markup Language (KML) format. An even more advanced diversion of these services is called WFS-T, which helps not only looking-up but also manage (INSERT, UPDATE, DELETE) spatial data at a feature level, in addition to VIEWING and QUERYING. Some valid request performable by WFS server embraces:
  - **GetCapabilities** – returns XML parameters about the list of WFS data, operations and parameters
  - **DescribeFeatureType** – returns information and attributes about a specified WFS layer
  - **GetFeature** – returns the data itself in the form of geometry and attribute values
  - **LockFeature** – guards a given feature from being modified

There are also some other batches of standard spatial Web services which are said to be not so important for this project. But, to make the list complete, let us have a very shallow look at them.


- **Web Catalog Services (CSW)** - semantic searching of metadata related to spatial data and services[21]

- **Web Processing Services (WPS)** - requesting the execution of client- or server-side process on spatial data[25]
4.3 Enhancing MVC spatially

Originally, Web frameworks are not distinctively meant to target the spatial Web. They first set an eye on how to quickly develop most commonly expended non-spatial Web applications. Although we priorly said that spatial data are intruding and are no longer rare around the Web, our scrutiny shows initiatives to expand the realm of the Web frameworks into the geographic Web are almost few, if not nonexistent. Incidentally, GeoDjango’s endeavor is repeatedly cited as the leading and regimented movement towards spatialization of a MVC-led Web framework (the Django).

However, an even worser situation, such initiatives are not only petite in scale but also independent in the sense that there is no universal rule that guide them. This section of the chapter attempts to develop a guideline on how to spatialize MVC-led Web frameworks, footing on spatial Web-database practices and technologies in place. This in turn enables our forthcoming discussion on Web framework’s spatial database consistency to have a spatial base, which is why we spatialized web2py based on the newly designed spatialization guideline.

4.3.1 Learning from pioneers

GeoDjango, flagged as the path-finder in taking the first initiative to exploit Web frameworks for spatially-aware Web applications, is a geographic limb of Django that takes care of spatial capabilities by mainly extending the Django M (in MVC) with a field of data type (GEOMETRY). This way, its set-up was restructured to let it edit, insert, look-up and display spatial data on the Web. There is a support for spatial data with the following geometry types: PointField, LineStringField, PolygonField, MultiPointField, MultiLineStringField, MultiPolygonField, GeometryCollectionField. All of these types correspond the geometry classes modeled by OGC® simple feature specifications[70], illustrated in Figure 3.2, Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, GeometryCollection, respectively.

GeoDjango, included within django, houses extensional modules and classes that are able to deploy geo-oriented Web. In addition to the standard python and django libraries, full-fledged geospatial application development using GeoDjango requires two other classes of open source libraries:

- Spatial database - the database back-ends supported by GeoDjango are four: PostGIS-extended
Chapter 4. Spatialization of the Web Frameworks

PostgreSQL, SpatiaLite-extended SQLite, Oracle and MySQL

- Spatial libraries:
  - GEOS\cite{91} - a C++ version of JTS (Java Topology Suite), a library with added functionalities to perform geometric operations (using python built-in API, ctypes), and in compliance with the OGC\textsuperscript{R} simple feature specifications for SQL\cite{70} spatial functions and operators.
  - PROJ.4\cite{92} - a cartographic projection library to switch between Spatial Reference Systems (SRS) that define the spatial data, a package used together with PostgreSQL’s PostGIS and SQLite’s SpatiaLite
  - Geospatial Data Abstraction Library (GDAL)\cite{36} - an abstract data model translator library to read, import and export (using python built-in API, ctypes) most vector and raster spatial data formats, hosting OGR Simple Feature Library\cite{69} which GeoDjango uses for vector capability
  - IP-based Geolocation library (GeoIP) - an optional C library to query some location-based information drawn from an Internet Protocol (IP) address

To add more on this subject, it is commendable to point out here that the three open-source packs (GEOS, PROJ.4 and GDAL) are part of the latest releases of spatial databases (including PostGIS). There is no need of treating them separately. From now on, the reader should keep in mind that all references to a spatial databases indirectly implies the inclusion of these open libraries, at least for our discussion.

GeoDjango supports a set of matured data formats. The set includes the followings: Google’s KML, OGC’s GML and WKT, GeoJSON, a geo-extension to JavaScript Object Notation (JSON)) and GeoRSS (geographic version of a Web feed, Really Simple Syndication (RSS)). An integration of Openlayers API and Google maps API with an implementation of geodjango is also possible. The simplified architecture of geodjango is presented in Figure 4.3.

![Figure 4.3: Overall architecture of GeoDjango](image)

There might be several ways to carry out a spatial enhancement of Web frameworks. Our recommended approach is to follow the customary MVC categorization and treat each piece separately as what they require for spatial-enhancement varies accordingly. This guideline is needed to be reproducible, hence first a more generic discussion precedes its particular examination on web2py. The whole process has been summarized in Figure 4.4 and its detail is presented as follows.
4.3. Enhancing MVC spatially

Figure 4.4: Spatial extensions to non-spatial tables
4.3.2 Extending the M (of MVC) in Web frameworks

The C of CRUD

Models (M of MVC) in established non-spatial Web frameworks are capable of creating and manipulating tables that have no spatial behavior. To make them also capable of dealing with spatial tables, their current modeling library should be mutated (such as the DAL in web2py, that holds all database-allied classes and methods). Meaning, a stratum that assimilates basic geometric primitives and principles should be added into it. This section narrates the basic concepts in spatial databases in a way it is possible to develop a guideline on ‘how-to?’ create spatially-aware M of Web frameworks. Discussion about its implementation on web2py follows it. For the most part, it involves the ‘CREATE’ constituent of the CRUD (discussed in Section 1.1.2) capability of databases.

Prerequisites

Spatial enablement of relational tables means making it hold at least one column that defines and manipulates location-based data. This added column is a special type of column in some demeanor. One of its unique characteristics, for example, is it defines a totally different version of data field than other alphanumeric data types, which conceptually is called GEOMETRY. It assumes a collection of point-set data, as discussed in Chapter 3.

Below is a précis of standard-based principles of affixing geometric field into non-spatial tables. Exclusively, details about addition and manipulation of geometric tables in PostgreSQL can be found in the online PostGIS documentation[58]. The discussion exemplifies the PostGIS spatial database, introduced in Section 3.8. Our focus on PostGIS is only a matter of choice, there are many options. However, the underlying concept remains the same, no matter which database server is picked.

The creation of all-inclusive geometrically-aware PostGIS database gives rise to an imperative requirement, it should be pre-equipped with ‘spatial_ref_sys’ and ‘geometry_column’ tables. These are the two mandatory tables required for a fully functional PostGIS spatial databases. The ‘spatial_ref_sys’ table is a store of over 3000 branded spatial reference systems with the following five attributes:

- **Spatial Reference system Identifier (SRID)** – the primary key that identifies each spatial coordinates system stored
- **auth_name** – the standard responsible body for the spatial reference system stored (e.g., EPSG, which stands for European Petroleum Survey Group)
- **auth_srid** – the SRID exactly as defined by the responsible authority
- **srtext** – The WKT equivalent of the SRS
- **proj4text** – a Proj4 string that defines the SRID (Proj4, refer Section 4.3.1)

The second requirement, the ‘geometry_column’, is a table with the following attributes that automatically keeps a record of details about each geometry columns introduced into any table defined in the database. Such details are appended into the geometry_column table on addition of any geometry column. An upcoming discussion shows how are geometric columns added into a PostgreSQL table.

- **f_table_schema** – the name that indicates where in the database schema the geometry column has been created
- **f_table_name** – the name that indicates in which table the geometry column has been added
- **f_geometry_column** – name given to the geometry column itself
- **coord_dimension** – the geometric dimension specified for the geometric column
- **srid** – the SRID specified for the geometric column
4.3. Enhancing MVC spatially

Creation of a spatial table

As a general rule, the formation of a PostGIS spatial table involves two basic steps. Firstly, the non-spatial columns of the table are defined customarily by using the ‘CREATE TABLE’ SQL statement, which among other things asserts each attributes and their data types. This step involves purely PostgreSQL dialect and needs no special dealing. The following line delineates the basic syntax:

Basic syntax:
CREATE TABLE table-name([attributename datatype]);

E.g.:
CREATE TABLE interfluve (    fid INT,    fluvename VARCHAR(50),    inamazonia NOOLEAN);

Afterwards, the so called GEOMETRY column/s are defined and added to the defined columns by using the ‘ADDGEOMETRYCOLUMN’ statement native to PostGIS lingo. Its syntax looks and is defined like below:

Basic syntax:
SELECT ADDGEOMETRYCOLUMN([‘schema-name’], [‘table-name’], [‘geometry-column’], [‘SRID’], [‘geometry-type’], [Dimension]);

E.g.:
SELECT ADDGEOMETRYCOLUMN(    ‘spatialDB’, ‘interfluve’, ‘geom’, ‘4326’, ‘MULTIPOLYGON’, 2);

The above ‘ADDGEOMETRYCOLUMN’ SQL command assumes the following basic parameters:

- ‘schema-name’ – (a string of database schema name) is optional and can be skipped if we are sure about the current database schema we are working with (i.e., ‘spatialDB’ in the example)
- ‘table-name’ – a string of the table name onto which the geometry column is to be added (i.e., ‘interfluve’ in the example)
- ‘geometry-column’ – a string name of the geometry column to be added (i.e., ‘geom’ in the example)
- SRID – an key integer from ‘spatial_ref_sys’ that uniquely identifies the SRID. Default is -1 that indicates no-SRID (i.e., 4326 in the example, for World Geodetic System (WGS) 84 lat-lon reference system)
- ‘geometry-type’ – the geometry data type (any one of the low-level data types indicated by OGC® Simple Feature Hierarchical Model in Figure 3.2) (i.e., ‘MULTIPOLYGON’ in the example)
- Dimension - an integer indicating the geometric dimension, default is -1 to mean no dimension (i.e., 2 in the example, to mean two-dimensional)

In the long-run, while taking data to and from a spatial table, a serious inquiry about the performance issue of the look-up certainly will raise as geographic data are known to be colossal in size and costy in computation. Hence, spatial databases use spatial indexes to effectively optimize the spatial querying plans. R-tree, Octree, Quadtree and GiST are few of the common options available for spatial database. Generalized Search Tree (GiST) is generic type of indexing preferred by PostGIS. The following line adds it to a defined spatial table:
Basic syntax:
CREATE INDEX [unique-indexname] ON [table-name] USING GIST ([geometry-column]);

e.g.:
CREATE INDEX interfluves_gist ON interfluves USING GIST (geom);

In the process of adding a spatial column to a table, PostGIS implicitly adds three important check constraints (see what check constraints are in Section 3.6.1). They are consistently given the names ‘enforce_srid_[geometrycolumn]’, ‘enforce_geotype_[geometrycolumn]’ and ‘enforce_dims_[geometrycolumn]’ by default. They set constraints that watch and block inserts/updates other than the indicated geometry type, SRID and dimension, respectively. For instance, in the above example, a geometry of type ‘POINT’ would be considered invalid and rejected from being admitted to the database.

Having said this, we now arrived to the point where we discern the line-of-attack to introduce spatial behavior into Web frameworks’ table object. As a result, we dare to say that the first step in spatializing web frameworks is to extend their ‘Table’ object by defining a new ‘Field’ object that spatially enables it. Plainly, this means an addition of a new field object that adds geometry column/s to an instance of table object, the same way discussed above. That’s why we recommend a conception of an abstraction layer and its associated functional classes that instantiate new geometry field (and possible operations) to the existing table object inside the Web framework’s library.

At this point, it is important to highlight the important steps which should be identified before hand. Any one who is interested in spatializing the M of MVC can indeed follow these steps. Their definition is not particular to any specific implementation. To test how it works, the following discussion follows these steps in an attempt to spatialize the M of web2py:

1. **Step 1**: Define target geometry structure to be supported
2. **Step 2**: Create a callable Wrapper that instantiates a spatially enabled column
3. **Step 3**: Create another internal Wrapper that materializes the instantiated spatial column, by means of existing Database connector

### Extending the M of web2py

Before we dive into the spatial extension we created to web2py, we should first uncover the existing table, column and database connectivity creators inside it. In web2py, every communication of the any of the MVC components (especially, the Model) is an instance of the connection wrapper (called SQLDB) that defines database connectivity. This object requires a string specification of the basic connection parameters: `database server name`, `user-name`, `password`, `host`, `port` and `name`. What appears below is a typical case of web2py’s connectivity to PostgreSQL server:

Basic syntax:
```python
q=SQLDB('postgres://abc:xyz@localhost:5432/spatialDB')
```

This line of code, usually the first thing to do in the Model, assigns the connection object to the variable q and make it known not only in the Model environment but also in the View and Controller. For simplicity, it is splittable into the following constituents:

- **q** – the variable that assumes the connection
- **SQLDB** – a web2py wrapper for database connectivity
- **postgres** – the database server name
- **abc** – valid username as a doorway into the database
- **xyz** – password specification of the username indicated
- **localhost** – host of the database server
- **5432** – port number of the database server
4.3. Enhancing MVC spatially

- **spatialDB** – name of the spatial database

Executing any web2py operation into the database must be enclosed by an instance of the declared connectivity. For instance the ‘define_table’ function is a model-level wrapper that gives birth to ‘CREATE TABLE’ SQL command to create a new table inside the database. A request to run it on the database requires setting of the connection object ‘q’ first, as it appears below:

Basic syntax:
```
q.define_table('[table-name]',
               SQLField('[attribute-name]', [datatype])
)
```

E.g.:
```
q.define_table('interfluves',
               SQLField('fid', 'integer'),
               SQLField('fluvename'),
               SQLField('inamazonia', 'boolean')
)
```

The underlying notion is identical to the ‘CREATE TABLE’ case we discussed above. It creates new table object (‘SQLTable’ in web2py) with the name of ‘interfluves’ and the ‘SQLField’ wrapper creates the necessary attributes of the specified table. Each of these object can take more optional arguments which we did not include here for now. But, in defining a ‘SQLField’, a string of the attribute type should at least follow each attribute name, preceded by comma. If otherwise, it means the data is defaulted to ‘string’ (equivalent to ‘TEXT’ or ‘VARCHAR(n)’ in PostgreSQL, the n inside the bracket being the maximum character length allowed). That means, column name is the only mandatory argument of an instance of ‘SQLField’ object.

Now, let us come to our specific case. No single geometry data type (see Figure 3.2) is known to many Web frameworks column definers. The same actuality also pertains to web2py’s SQLField. We proposed a nonspecific guideline to spatialize the non-spatial data abstraction layer of existing Web frameworks, based on the basic principles instigated above (as shown in Figure 4.4). The design considers the state-of-the-art practices and tools which are prominently cultured around spatial Web.

By following the spatilization guideline and the steps identified above, we specifically made an extension of the basic elements of the M of web2py to bring it into the realm of spatial world. The first step was to widen the set of supported data types inside the web2py data definer layer. As it is shown in Section A.1, we decide on the inclusion of the six top-simple geometry data types exposed by OGC®: POINT, MULTIPONT, LINESTRING, MULTILINESTRING, POLYGON and MULTIPOLYGON. This made web2py become acquainted with spatial data types.

Addition of new data type alone does not at all realize the spatial edition of a Web framework. Thus, the next step was to make the existing table and column constructors know and handle them. New column definer that handles geo-data type was created with the name gField, specifically inside the ‘DAL’ segment of web2py (the layer that defines Table/Field objects and operations within a database connectivity). This is shown in Section A.2.

The third and physical step is to put an internal wrapper. What makes it internal is that it is hidden from the application development layer. It only realizes the creation of the gField object defined. What users specify and know is the gField as wrapper to make a definition of the geometry field of their interest. The internal Wrapper should take care of the rest - by making use of the database connector, it adds the geometry column into the required table (see Section A.3). This augments the ability of web2py a step higher by making it create standard-based geometric data type. Let us see an example on how to use the new gField wrapper as geometric column adder.

Basic syntax:
```
gField([attribute-name], [data-type], [SRID], [dim], [index])
```

E.g.:
As can be seen in the example above, `gField`'s use on the development side is compatible to the existing ‘SQLField’ object by following the same syntactic structure. Other than the two types of parameter it shares with the ‘SQLField’ (i.e., attribute-name and data-type inputs), the `gField` object takes the following three more defaulted inputs:

- **SRID** – the spatial reference id (default is 4326)
- **dim** – geometric dimension (default is 2)
- **index** – (True or False) whether GiST spatial index should be created or not (default is True)

### 4.3.3 Extending the C (of MVC) in Web frameworks

**RUD of CRUD**

When we say spatial columns of a relation are unique and different than non-spatial ones, it is not only in the eyes of the way they are created but also the way they are operated. This is to mean the spatial columns require not only unique DDL statements but also DML statements. In Web applications that require database transactions, SELECT, INSERT/UPDATE and DELETE are the most busiest SQL requests to make manipulations of the stored data. These cover the ‘RUD’ operations of the ‘CRUD’ database functionality.

Our discussion here also follows the approach used above in our discussion about extending the M component. First, the general RUD operations of our example database, the PostgreSQL, are discussed. Hence, insertion and updating of attributes of relations take the following SQL syntax:

**Basic syntax:**

```
INSERT INTO [table-name] ([comma separated list of table attributes])
VALUES ([comma separated list of attribute values]);
```

e.g.:

```
INSERT INTO interfluves (fid, fluvename, inamazonia)
VALUES (21, 'abc', 'xyz');
```

**Basic syntax:**

```
UPDATE [table-name]
SET [comma-separate-list-of-attribute-values-pairs]
WHERE [condition];
```

e.g.:

```
UPDATE interfluves
SET fluvename='efg'
WHERE fid=21;
```

A variety of vector formats exist that are fully supported by spatial databases, such as PostGIS, part of which are the followings: WKT, GML2 and GML3, **Geographich JavaScript Object Notation (GeoJSON)**, **Geographic Really Simple Syndication (GeoRSS)** and **Google’s KML**. Among them, WKT is a ‘well-known’ textual human readable markup language that represents vector geometry, with its
4.3. Enhancing MVC spatially

binary machine-readable equivalent known as **Well-Known Binary** (WKB). Most commonly, almost all spatial databases contain functions that vector geometries to and from WKT and WKB. In PostgreSQL, pgAdmin III can be used to interact with the database and execute such functions. See the examples below.

Assuming our example geometry exists in a WKT format, the following example shows that geometric INSERT/UPDATE assumes a slightly different way of assigning geometric values to its corresponding column:

Basic syntax:
```
geometry_column = ST_GeomFromText('WKT', SRID)
```

* e.g.1:
```
```
```
e.g.2:
```
UPDATE interfluves
SET geom=ST_GeomFromText(...)
WHERE fid=21;
```

To understand what is happening here better, an explanation of the above INSERT/UPDATE elements follows:

- **geometry_column** – name of the geometry column that assumes the insert/update
- **ST_GeomFromText** – a geometry object creator that inputs a Well-known text representation and its SRID
- ‘WKT’ – a human-readable string of the geometry data as set of values of each node
SRID – the spatial reference id used

Having this in mind, it is rational if one say the overall geometric INSERT/UPDATE SQL syntax is not much different from the ordinary way except that the messy geometric attribute value is wrapped by geometric operator that interprets it before it is loaded it to the database, as in the examples above. The remaining CRUD operators, SELECT and DELETE commands do not make any difference on both spatial and non-spatial tables.

Basic syntax:

```
SELECT [comma-separated-attribute-names or valid SQL column functions]
FROM [table-name] AS [alias]
WHERE [condition] [e.g., id=1];
```

--By valid SQL column functions, we mean SQL functions that formats the attributes such as \( \text{SUM()} \) and \( \text{ROUND()} \) that return the a summed-up and rounded-off values of specified attribute/s.

E.g.:

```
SELECT * FROM interfluves WHERE fid=21
```

--retrieves everything (*) from interfluves table whose fid are 21

```
DELETE FROM [table-name] WHERE [condition] [e.g., id=1];
```

--condition limits rows of records for which the operation is expected to perform.

E.g.,

```
DELETE FROM interfluves WHERE fid=21
```

--deletes a row in interfluves table whose fid is 21

One of the advantages of these RUD operators is that developers/users can exploit them as integral part of their complex, application-specific server-side functions. Almost all databases offer the ability to create stand-alone functions, similar to our discussion in Section 3.6.1. Such functions can include any combination of the above straightforward SQL’s to produce a database procedures or methods which are highly complex, advanced and better, because they can be made to better suit an application domains.

Having explained simple examples of valid SQL statements to communicate with databases manipulating stored tables, attributes and tuples, the next question would be how to make a spatially enabled C in MVC-led Web frameworks. As Figure 4.4 depicts, we proposed below two options of spatializing the C.

1. Create a wrapper on top of existing table manipulators in order to properly handle the variances
2. Create a wrapper that gives access of server-side application-specific functions, (also called callable functions in Section 3.6.1)

The first case, seems highly straightforward and immediate solution. It requires the augmentation of existing data manipulators inside the Web framework (such as the insert, update, delete, select and etc in web2py) to re-create them into an advanced wrappers that incorporate and resolve the small discrepancy that emanate due to the introduction of a spatial behavior. However, our recommendation is for the second option; creation of a wrapper that gives easy access and use of server-side functions; on a self-contained spatial database. The justification to this is that it is found to be eXtensible and suitable for application-specific operations. The second and more convincing reason could be the advantage it gives to our new architecture defined in Section 3.7, to enable us handle beyond-the-subset (as defined in Section 1.2) consistency rules which are still application-dependent.

**Extending the C (of MVC) of web2py**

To discuss about the extension of the C using the second recommended option above, the question to pose is how to make a call of user-defined server-side functions from the controller. In web2py, there is no
4.4 Conclusion

ready-made way of calling database functions from the development layer. Preparing this function-caller wrapper required the inclusion of few lines of code inside the web2py (see Section A.4).

We baptized the wrapper that invokes server-side functions as fire. It is an instance of the database connectivity and takes a string of the signature of the to-be called function as an argument. Also, it returns the value the called database function returns. Its usage looks like as it is shown below (‘q’ being the database connectivity). Its tangible utilization is by an extended controller, inside actions that call as much application-dependent functions as possible (for example, see the Update controller action that inserts/updates data into distribution and species tables in Section 6.4):

\[
\text{q.fire([function_signature])}
\]

4.3.4 Extending the V (of MVC) in Web frameworks

There are many rich and potential open-source libraries that are capable of spatially-enriching the view component of Web frameworks. Figure 4.4 OpenLayers API, Google Earth API and Mapnik are some of the widely used such libraries. The first one is the most famous and commonly used source around spatial Web environment to add map interfaces with vital interactivity. It is an open-source JavaScript mapping library for building rich Web-based standard spatial applications, such as the OGC® industry-based WMS and WFS protocols. It has been successfully integrated into web2py and implemented in the WFS-T prototype developed in Chapter 6 (See Section 6.3.2). For complete inspection of the API, consult the ultimate documentation in [4].

4.4 Conclusion

In the present chapter, we tried to sketch out fundamental conceptions to answer the question, ‘What gives the Web a spatial behavior and how to inherit that into Web frameworks?’. Section 4.2 introduced the notion of spatial Web in the eyes of standards and technologies helpful for the materialization of location-based Web services. In the same section, following the elucidation of the famous SOA, we introduced the concept of standard OGC® led spatial Web services. The later seems quite fresh to Web frameworks engineering environment.

In view of that, Section 4.3 pushed the geographic protocols and practices of the Web into and to occupy the territory of Web frameworks. We learned important lessons from the senior spatial Web framework, GeoDjango, to discern how it devised its architecture to import practices, tools and technologies from the spatial Web. Then we got hold of a simplified and directive configuration about the enhancements which should be made to MVC constituents separately. We abstracted this configuration in Figure 4.4. Chasing this direction, the guideline was examined for its workability on Web2py, by finally creating a spatially enhanced edition of web2py. The enhancement itself was tested in the yet to come Chapter 6, where we stabbed to build the first standard spatial Web application using this fresh release.
Chapter 5

The design process: design-driven modeling

5.1 Introduction

This chapter overviews a use case for a test of consistency-preserving spatial database usable by Web frameworks, referring web2py as a representative. As a matter of focus, the to-be built Web-accessible application should display some spatial (vector) data functionality and demonstrate a consistent interaction privileged (registered) users make with the back-end.

The design process we are following here was meant to be reproducible, because it is based on the generic design frame adopted in Section 2.5. The procedures can be followed every time a user comes with different use-cases. UWE being our Web engineering approach of choice, the description starts from Section 5.2 by first conceptualizing the agents and their role, together with the system functionality our Web-based application is expected to trigger interactions with the spatial database beneath it. This makes the CIM. The section that follows (Section 5.3) describes textually and diagrammatically the PIM of the information content, content modeling. Please be noticed that, Section 2.5 argues the Navigation and Presentation models are not part of design-driven process because they are found to be less important for our topic. The PSM is considered as part of the design process but, it was separately covered in Chapter 6 as it incorporates many implementation details. The conclusion section (Section 5.4) ends the chapter.

Most important attention was given in the design process to include the specification and enforcement of constraints in all relevant phases. We tried to make the list of constraints complete enough to give a demonstration of specification and enforcement ways at every required stages of the process. This list is by no means conclusive.

5.2 Requirements Engineering: the CIM

This use-case is about a working project that has been running, slowly, since 2006. The project runs collaboratively with INPA, Manaus Brazil on an information platform about Amazonian life forms, i.e., plants and animals and their distribution. At present, only avian life forms are covered, for one because for birds distributions are on average much better known than for other life forms. But over time, the coverage is expected to be extended to aquatic animals like crustaceans, but also mammals, and plants.

The Web system is expected to capture information about species, specifically names in three languages as well as spatial distribution for most (but not all) species. The species names would be captured in different forms; listname is the scientific name, port_ and eng_name give Portuguese and English vernaculars. For historic and archival reasons, spatial distributions are modeled as separated objects. The default SRID for the distribution and other geometries in the domain is 4326. In addition, each species should keep a record of other three attributes; a unique identifier, a name of legal authorized body and taxonomic rank.

Also, the system should maintain separate objects for avian genera and avian families, with the obvious biotaxonomic structure in place. It also maintains a single spatial object, which is a Brazilian
legal entity, known as Amazonia Legal, that demarcates the legal sphere of influence of the the authorized body. This area enjoys special protective measures under Brazilian law, for instance laws about the lands of indigenous people, laws about deforestation, and so on. The purpose of our system is to maintain as faithfully as possible the avian distributions known for the area of Amazonia Legal.

Supporting information comes from interfluves and ecological regions. An interfluve literally is an area between rivers, and for the purpose of mapping species’ distributions, Amazonia Legal has been ‘cut up’ into interfluves. Interfluves are a biogeographic reality in the sense that their borders (the rivers’ catchment areas) form natural boundaries for many species. It was later decided to expand the notion of interfluve to the neighboring areas of Amazonia Legal, both inside and outside of Brazil. Interfluves have a name, a polygon geometry and an indication of whether they are inside Amazonia Legal or not.

![Figure 5.1: Use case illustration](image)

### 5.2.1 User requirements

There are two types of users, anonymous and registered (see Figure 5.1). By anonymous we mean those who are recognized by the system as less-privileged to make change to data content, hence can only navigate along the interactive interface and make querying (see ‘viewMap’ and ‘queryData’ use-cases in Figure 5.1). If they are given full mandate of introducing data changes, they take the name ‘registered users’. Which means, they can run an application that manipulates stored information in the database (see the ‘updatData’ in Figure 5.1). The identification of users is based on user-name authentication. Users should be requested for logging-in if they have private identifier. Or, they could be asked for signing-up (see the ‘Sign-up’ use-case in Figure 5.1); new unique user-names can be issued by validating users input credentials. The ‘Sign-up’ use-case can be extended to ‘Login’ use-case, meaning users who are confirmed to qualify for an issuance of a user-name would be redirected to the ‘Login’ page (see the «extend» relationship between the ‘Sign-up’ and ‘Login’ use-cases in Figure 5.1, see also the discussion about «extend» relationship in Sub-section 2.5.1).

We differentiate between two types of users to make the system more complete. Nevertheless, our priority here is on the actions performable by registered users, including database administrators.
5.2.2 Functional requirements

The Web-based application that we have in mind here is essentially a content management system. The reason is that in such a system data calls of functions database functions are more prominent. We do describe all these functions as integrity-preserving functions, i.e., they are meant to never invalidate database integrity and an implicit pre- and postcondition is always that the database is consistent. Hence, it should be implemented as such.

Below are seven server-side functions that our Web application should offer to its users. These are the kind of functions which are called **callable functions** in our constraints enforcement discussion in Section 3.6 and implemented in Section 4. They are grouped as update functions in general, because they target at the modification of different contents of the database (see ‘updateData’ use-case in Figure 5.1, see also the discussion about the ‘Generalization’ relationship in Sub-section 2.5.1). Only authorized (registered) users can make modifications to data from the Web interface by triggering these modifiers. The ‘viewMap’ use-case is part of the ‘updateData’ use-case, which in turn includes the ‘queryData’ use-case (see the two «include» relationships in Figure 5.1, see also the discussion about «include» in Sub-section 2.5.1).

As described in item 3 of Section 3.6.1, the philosophy on functional development of these update use-cases is to include them as much as possible as stored procedures in the database, so as to make the database a self-contained and functionally complete resource. In the application code outside the database, each of them would be visible as a single function call, hence the name callable. This contrasts to an approach in which much of the database object logic is encoded in the Web frameworks application. However, the newly redesigned architectural set-up in Chapter 6 resolves this situation for the sake of enhanced database consistency.

Add distribution geometry

The Web system should offer an application that enables authorized users insert new data about species distribution. The geometric data can be obtained from an indicated shapefile, and append it to the existing distribution. This is a first version of the geometry addition. A second version of this function should allow on-screen digitization of the geometry. In both cases, the stored geometry should remain simple and valid. Plus, it should fall inside the bounding box that envelopes Amazonia legal at some units of buffer zoning.

The application logic in the first version of the geometry insert is totally dissimilar. The system should give the user an opportunity to locate the to-be appended spatial dataset. Calling a database function to push this dataset into the database seems unfeasible, but the system should afford a realistic option like making use of add-in tools that load spatial datasets onto spatial database (For instance, shp2pgsql.exe and psql.exe in PostGIS). This extra version is something an implementor might decide at will.

In the later version, we named the callable database function that performs the event ‘add_distrib’ (see ‘add_distrib’ use-case in Figure 5.1). The Web application flow looks like the Activity diagram in Figure 5.2. It starts by picking an on-screen digitization tool that enables the Web user draw the geometries for ‘INSERT’. To make the geometries usable arguments for callable database functions, they should be serialized into a format readable by the system, usually into ‘WKT’ format (see the discussion on vector formats in Section 6.3.2 of Chapter 6). Next, the application knocks the database for ‘INSERT’ command, to store it as species distribution data. The database’s reaction in the form of ‘Success’ or ‘Error’ steps next. The message should be explicitly reported back to the Web application in order to inform the user with what happened beneath the application.

Update distribution geometry

This is another looked-for callable database functions that manipulates the species distribution geometry data. The first thing the user needs to do in this case is to toggle existing species distribution geometry and make needed changes to it. Plus, it should not be skipped that the same geometry should be serialized into a vector format to make it readable by the callable functions. The function must also be accompanied by another action that collects inputs for the modification of the non-geometric behaviors of the species whose geometry is being updated. All constraints applicable to the insert case above are inheritable into this function too. Successful execution of the update transaction should let the database send a ‘Success’
Figure 5.2: Activity diagram: depicts structure of add_distrib callable function
message, a ‘Failure’ otherwise. The name given to this callable database function is ‘update_distrib’ (see ‘update_distrib’ use-case in Figure 5.1). The Activity diagram in Figure 5.3 points up the flow of the update geometry action.

![Activity diagram: depicts structure of update_distrib callable function](image)

**Figure 5.3: Activity diagram: depicts structure of update_distrib callable function**

### Delete distribution geometry

In spatial databases, deletion of geometric features is also considered as important as insert and update are. We also seek the Web system to have a tool in place that enables users remove distribution data from the database. The most central characteristic of data deletion in relational databases is the REFERENTIAL INTEGRITY constraints, which watches the deletion of a record that has a value referred to it by means of a FOREIGN KEY (see Section 3.4.2). Hence, removal of distribution geometry is possible if it is not violating the referential integrity constraint.

Usually, while an event to delete data begins, it is intelligent to warn the user for confirmation of the action. Our intended Web system should also do that. Once an approval was received, the to-be deleted geometry should be ready to be called by the callable database function. The sending of ‘Success’ or ‘Error’ messages back to the Web application layer are still likely. The database function that does this is called ‘delete_distrib’ (see ‘delete_distrib’ use-case in Figure 5.1) and its functional flow looks like Figure 5.4.

### Restrict distribution geometry

The meaning of this application is similar to the updating of the distribution geometry above (Section 5.2.2). However, in this case the source of the geometry for update is obtained by restricting the distribution of a given species to a given list of interfluves. As it is clearly illustrated in Figure 5.5, first, identification of input interfluval boundaries and the to-be restricted are indicated by their identifiers, the map interface in the Web app may provide a picklist or tool to select geometry features displayed in the
5.2. Requirements Engineering: the CIM

Figure 5.4: Activity diagram: depicts structure of delete_distrib callable function
map interface. Once the callable version of the geometry has been obtained, while an update command is sent to the database, the interfluvial boundaries identified should be combined into one geometry or all of them can be stored as a single geometry that accommodates multiple geometries. Sending either ‘Success Message!’ or ‘Error Message’, whichever applies, is still expected before the termination of the application logic. The callable database function assigned to accomplish this task take the name ‘restrict_distrib’ (see ‘restrict_distrib’ use-case in Figure 5.1).

![Activity diagram: depicts structure of restrict_distrib function](image)

**Figure 5.5: Activity diagram: depicts structure of restrict_distrib function**

### Add species data

The Web system should not only store the geographic characteristics of the bird species, but also other thematic behaviors. The complete list of attributes that should be considered to describe every species are included in the definition of the information content in Section 5.2.

To avoid some inconveniences, collection of species information might be done parallel to the insert/update of the species distribution data. The structure of this application is traced in Figure 5.6. An easy to use information collector form should be presented. Upon receipt of the completed forms, the Web app wraps them by the callable functions and sends the insert/update command into the database. The conditions in this case are two: first, new species information should get stored if there is no existing data taking the same name (in all languages); and, second, null value is not an option. If these conditions are met, ‘Success’ message is compulsory. Otherwise, ‘Error’ message should be received. Here, ‘add_species’ is the callable function (see ‘add_species’ use-case in Figure 5.1).

### Add family and genus data

The general flow of these two applications is not so much different. They meant to add family and genus entries into the database, respectively. And, they use the ‘add_family’ and ‘add_genus’ database functions (see ‘add_family’ and ‘add_genus’ use-cases in Figure 5.1). The overall flow of their application logic is
Figure 5.6: Activity diagram: depicts structure of add_species function
seen in the Activity diagram shown in Figure 5.7. Right after the start of the application logic, a form that collects the input data pops up. The alphanumeric data feed by the user would be wrapped with the pertinent caller. The database function should make a decision whether an entry is ‘INSERT’, an action to store them for the first time, or ‘UPDATE’, another action to make a modification of existing entry.

The transaction should be checked for inconsistency to enforce some valuable constraints, such as the uniqueness of the names. Before the ending of the application flow, an integrity/consistency report should be sent to the Web application. As usual, such a report is either a ‘Success Message!’ upon successful execution of the transaction or an ‘Error Message!’ upon detection of any sort of violation of a specified constraint rule.

**Figure 5.7: Activity diagram: depicts structure of add_family and add_genus functions**

### 5.2.3 Constraints identification

The above use-case description holds a set of constraints. At CIM level, a list of four of them that are applicable in our forthcoming discussions about constraints specification and enforcement is presented below (see the Constraint notations used in Figure 5.1). This is not a complete list in the sense that the main intention is only to come up with an enumeration of constraints that are complete enough to make a spatial database consistency test to Web frameworks, at least one at each level like our discussion in Section 5.3.1.

- **Constraint1** = Family and genus names should be unique
- **Constraint2** = All names of species should not be null and should remain unique
- **Constraint3** = Distribution geometry should be valid, closed and simple
- **Constraint4** = Distribution geometry should fall within the bounding box of Amazonia Legal (each distribution entry should be checked against the bounding box geometry of the Amazonia legal)
5.3 Content Model: PIM

In UWE, next to the system specification in the form of use-case and activity diagrams of the CIM phase, the PIM of the three levels of Web database application pursues. But, here, we are considering the content compartment only and we do not bump into the design issue of the other two levels. Figure 5.8 depicts simplified UML-based design of our spatial database. It is a class diagram of our use-case, plotted in EA project (see details about content modeling in Section 2.5.2).

The main aim here is to capture the main classes of the system in terms of content together with their attributes and inter-associations. We sorted out six classes of content; Species, Distribution, Genus, Family, AmazoniaLegal and Interfluves. To find out what they really represent, read their description provided in Section 5.2. The classes and attributes would be later, at PSM phase, mapped onto tables and columns. Attribute types must also be replaced by equivalent data type of the chosen database platform. More on that, the associations attached with between classes would be transformed into set of primary and foreign key relationships.

The relationship between the defined classes is a crucial extra information in the conceptualization of spatial databases. Above all, they hold constraints stored in the form of multiplicity. Like for example, a species may or may not (0..1) have a single distribution record associated with it, whereas a distribution have to represent one and only one (1) species record. The same holds for the relationship between Species and Genus: a genus may or may not be associated with many species records (0..*), and, on the other hand, species should have at least and at most a genus record associated with it. A genus could only assume one family class whereas a family should assume at least one genus, does not matter how many families to the maximum fall within the same genus class. However, the treatment of all constraints that emanate from the relationships between classes would be easier said than done and are not the main purpose of this project.
5.3.1 Constraints specification

We indicated that the main feature we want our Web application development process to take care of is the evident handling of constraints as integral part of the database schema, as it should be. For that, constraints should be explicitly defined in a way we can ensure their management. As a matter of practices native to database technology, they should be treated under three levels of abstraction, namely; object, table and database levels (based on our discussion in Section 3.6). All actions of the Web system should be deemed invalid if they don’t obey to the rules at hand.

We identified CIM-level constraints in Section 5.2.3 above. Their classification at different abstraction levels appears below. But, before that we need to enclose three more useful constraints that were not captured at CIM-level.

- Constraint5 = For associations between relations, their referential integrity should be preserved (example, between genus and family)
- Constraint6 = Each table should have uniquely identifier column (as discussed in Section 3.4.2)
- Constraint7 = Table species should contain unique combination of the names in three language

Constraints at object level

Once more, these are the constraints that are applicable to only a single attribute of a given instance. These may apply to one or multiple instances of a given relation, as long as the constraints can be checked at individual attribute basis. Technically speaking, such constraints can be seen at a conceptual level (attribute constraints) or logical level (tuple constraints) of the database schema. Our Web application is expected to keep an eye on following object level constraints (This is a simplified categorization of them, for a detailed discussion about the nature of such constraints, the reader is referred to a discussion on constraints taxonomy section, Section 3.4):

- PRIMARY KEY (Entity integrity rules) – Constraint6, see item 5.3.1
- NOT NULL (Attribute structural constraints) – Constraint2, see item 5.2.3
- Record- or tuple-level spatial constraints – Constraint3, see item 5.2.3

Constraints at table level

One-level higher than object constraints, we find constraints that must be checked at a level where at least two attributes are involved within a relation. This means, two or more attributes within a relation are checked for consistency before/after an insert/update of individual records is made. Here follows the constraint checks that involves more than a mere object.

- PRIMARY KEY (Entity integrity rules, COMPOSITE PRIMARY KEY) – Constraint7, see item 5.3.1
- UNIQUE tuple (Global, Entity integrity rules) - Constraint1 and Constraint2, see item 5.2.3

Constraints at database level

Equally important is the consistency check applicable at a wider view of the database. Any database functionality that involves two/more attributes of two/more tables require a different level of treatment than the above two. Here, the constraint checks go beyond a mere table boundary to guarantee a high-level consistency of the spatial database. Our case also involves such constraints, which are shown below:

- FOREIGN KEY (Referential integrity constraints) - Constraint5, see item 5.3.1
- Semantic Constraint – Constraint4, see item 5.2.3
5.3.2 Constraints enforcement

In the following chapter, an enforcement of these identified and specified constraints in a manner which we believe is suitable. The ways to enforce these constraints would be based on a categorization following the definition of level of complexity in Section 3.6. There is no one-to-one mapping between the levels of specification and enforcement. The decision on how enforce them depends on how complex is their computational algorithm. Sometimes, there is no need of creating more complication if they can be handled with the help of a simpler way. But, the decision is totally implementation and purpose dependent. The table below keeps a list of mappings possibilities between the possible constraint enforcement options available for each level of constraints specification.

<table>
<thead>
<tr>
<th>Level of constraint specification</th>
<th>Options of constraint enforcement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object-level</td>
<td>check, trigger and callable function</td>
</tr>
<tr>
<td>Table-level</td>
<td>check (very limited), trigger and callable function</td>
</tr>
<tr>
<td>Database-level</td>
<td>trigger and callable function</td>
</tr>
</tbody>
</table>

Based on the use-case outline in this chapter, our prototype development in Chapter 6 makes use of the most simplified available option for each case in point. For the constraints specified above, their preferred way of enforcement are shown in Table 5.2 below. This is only a choice; it is a decision dependent on the application and the implementor. The detail implementation entails in Section 6.2.1. One can notice here that many of the identified constraints were managed by using the simplest option available. Late addition of as much complex constraints as possible can be made by nesting them inside the callable functions that define the application logic (see Section 4.3.3), the highest-level option forwarded in Section 3.6.1.

<table>
<thead>
<tr>
<th>Specified constraint</th>
<th>Option of enforcement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint1</td>
<td>Legacy (null)</td>
</tr>
<tr>
<td>Constraint2</td>
<td>Legacy (null + unique)</td>
</tr>
<tr>
<td>Constraint3</td>
<td>Check</td>
</tr>
<tr>
<td>Constraint4</td>
<td>Trigger</td>
</tr>
<tr>
<td>Constraint5</td>
<td>check</td>
</tr>
<tr>
<td>Constraint6</td>
<td>Check</td>
</tr>
<tr>
<td>Constraint7</td>
<td>Check</td>
</tr>
</tbody>
</table>

5.4 Conclusion

The chapter we are finalizing were about design. It is an emanation of the concept of design-led modeling we introduced in Chapter 2, Section 2.5. It prioritizes the portion of the process that assembles the spatial database from more abstract into a concrete and physical version. Thus, 5.2 established the highly abstracted version of the Web-accessible database, which is far from the involvement of any computation. On the other hand, Section 5.3 covered the UML-led conceptual representation of the spatial database (the shift from the conceptual via logical to physical representation of the information content appears in the following Chapter). This was the section that tried to handle the inclusion of constraints as part of the database schema, following the new arrangement made in Section 3.7 of Chapter 3. It shows how constraints should be conceptually identified during the CIM phase and categorize them at PSM phase into classes that are believed to ease their own management. A rough mapping between the levels of constraints during specification and enforcement was also made.
Chapter 6

Implementation: Experiment the re-engineered method

6.1 Introduction

Following the use case characterization in the previous chapter, this chapter continues to reveal the necessary steps required to put them into practice. To manage and view geometric data stored in database, the most commonly used open-source thick clients are **User-friendly Desk-top Internet GIS (uDig)** and **Quantum GIS (qGIS)**. However, to fit the purpose of this research study, we found it convincing that the development of a thin client is appropriate than using thicker clients; which are still good options. Hence, in this chapter we developed a prototype based on our example use-case created in Chapter 5. We called this prototype a ‘CONSISTENT WFS-T prototype’. This web2py-powered Web application is unique in two ways. First, its being spatial (as a manager and viewer of stored geographic data) makes it a landmark in the history of web2py. Second, its being design-driven and consistent, makes it a landmark in the history of Web frameworks in general.

Here, the standard OGC-based WFS-T Web service was believed to be the optimal choice and building a WFS-T application using web2py is an integral part of our implementation. By this or by that reason, the condition is, the prototype should remain highly consistent unlike all existing WFS-T applications, and should follow the newly designed architectural proposal in Section 3.7. The prototype bases itself on example applications which exist in [5].

As you might expect, our discussion here also addresses the MVC compartments separately. Section 6.2 describes the M of our prototype, with basic constraints explicitly incorporated into the schema definition. It also narrows the situation into a web2py and PostGIS platforms, to finally materialize a functional spatial database. As we move on, we find Section 6.3, an implementation of the V of our prototype by making use of the wealth of functions included in the Open Source mapping library, the OpenLayers. The mediator python + web2py functions between the M and the V are covered in Section 6.4 that discussed about the C. Section 6.5 concludes the Chapter.

6.2 The M as a spatial database schema

The initial step was to design the database with the premise to not only represent the relations and their relationships, but also to specify the basic constraints that are believed to make the database stay very consistent. We already did that in Chapter 5.

The conceptualization in the content model (PIM, 5.8) shows that the information structure contains six model classes, each of which is treated as a relation/table at the PSM level ahead. And, attributes of each class will be mapped into columns, retaining the data type they had during conceptualization but equivalently local to the deployment platform. Association would be mapped into Primary and Foreign key relationships.

Transformation of the high-level UML-based conceptual schema, plotted in EA, into a logical model results, among other things, a structure that defines relation names, relationships, attributes, PRIMARY
6.2. The M as a spatial database schema

and FOREIGN keys (see Figure C.1 in Appendix C). This is still a PIM. It depicts the skeleton of our
data, without agonizing how to implement it. Further mapping of the standardized logical model onto
its equivalent PSM, PostgreSQL/PostGIS being our platform of choice, yields an intermediary text file
which comprises a set of CREATE TABLE and REFERENCES CONSTRAINT DDLs (see section C.2).

Section C.3 is the manually edited version of the physical model for database development specifically
on the chosen database platform. The manual edition of the output was required to make it directly
deployable on PostgreSQL (+PostGIS), as it includes some DDL concepts not applicable specifically to
the database platform of our choice and few inputs we don’t really need. Plus, to make a note of declared
constraints by the DDL or the incorporation of constraints that are not part of the transformation as
outlined in Section 5.3.1 of the design process.

Like for example, the declaration of UNIQUE PRIMARY KEY for some of the tables may or may not
be excluded for there is enough guarantee that web2py’s Model generates a unique identifier column (called
‘id’) to each tables defined. The following lines of code have been included into the table constructor
inside web2py, as part of the spatial enhancement. Thus, it gives the developer a possibility to declare
own PRIMARY KEY column, should he/she need it. See the example that follows it on how to make
use of it. The algorithm is straightforward. We first appended primary into the list of parameters of
SQLField object, the column constructor inside web2py. What it does is, it declares the corresponding
column as PRIMARY KEY (with the NOT NULL constraints attached to it), if the primary parameter
of the column has been set to ‘True’ (default is ‘False’). More on that, after web2py has already been
extended for the purpose this MSc project, the latest release of web2py already arrived with a more
advanced wrapper to define a KEYED table, with user-preferred PRIMARY KEY.

code:
if field.primary:
    ftype= ‘ PRIMARY KEY NOT NULL’

E.g.:
q.define_table(‘family’,
    SQLField(‘famname’),
    SQLField(‘fidx’, ‘integer’, True),
    SQLField(‘taxorder’, ‘short’))

Locate the ‘True’ value set to the ‘fidx’ column in the example above. You might have realized that
this value declares the corresponding column (fidx) serves as PRIMARY KEY of its parent table (family).

There are more things subject to edition. For instance, as it is exposed in chapter 4, the creation
of geometry columns does not follow the ordinary way of creating attributes of a relation as part of
CREATE TABLE command. Rather they apply a different syntax as is ‘ADDGEOMETRYCOLUMN’
specific to PostGIS, with table name, attribute name, SRID, geometry type and dimension arguments.
In our spatially-aware web2py, SRID and dimension should only be specified if they are different from
the defaulted values, 4326 and 2.

Also, we should infer from the CREATE TABLE SQLs that POLYGON (geom and buffer), MUL-
TIPOLYGON (distrib) and INT (taxorder) are the data types in PostgreSQL (+PostGIS) equivalent to
ST_MultiSurface, ST_Polygon, short, respectively. We might already have noticed the need for the decla-
ration of constraints that are not part of the model transformation, to enable the treatment of constraints
at schema level.

6.2.1 Web2py model

Once we come up with a designed model of our spatial database (with constraints explicitly defined), the
step that follows forward is mapping them onto a Web framework of preference (web2py, in our case).
This narrows the model into the M (to see how the M is constructed in web2py, consult Section 4.3.2).
It first opens the connection to the database server to execute the necessary DDL SQLs that it generates
at run-time (see Section C.5 of Appendix C). The generation is made by translating the web2py objects
into relational database’s standardized language (PostgreSQL + PostGIS, in our case), led by the ORM
principle (see Section 2.5.3). For the prototype at hand, the deployable code on web2py equivalent to the above PSM SQL code exists in Section C.4 of Appendix C. The constraint aspect of this web2py model is explained in Section 6.2.1 below. Table 6.1 depicts the basic ORMapping from PostgreSQL/PostGIS to web2py needed for the distribution table of our prototype, an example.

Table 6.1: ORMapping - PostgreSQL/PostGIS to web2py

<table>
<thead>
<tr>
<th>PostgreSQL/PostGIS</th>
<th>web2py</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONNECTION SQLDB</td>
<td></td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>q.define_table()</td>
</tr>
<tr>
<td>distribution</td>
<td>'distribution'</td>
</tr>
<tr>
<td>cidx INTEGER</td>
<td>SQLField('cidx', 'integer'),</td>
</tr>
<tr>
<td>SELECT ADDGEOMERTYCOLUMN('distribution',</td>
<td>gField('distrib', 'multipolygon')</td>
</tr>
<tr>
<td>'distrib', '4326', 'MULTIPOLYGON', 2);</td>
<td></td>
</tr>
</tbody>
</table>

Constraints specification as part of web2py model

For the sake of simplicity, the constraints that we want to enforce using web2py are categorized into levels, following our discussion in Section 3.4 of Chapter (see a simplified list of constraints of our use-case in the design process, Section 5.3.1). Plus, refer to Section 3.7 of Chapter 3 to see the newly projected re-engineered architecture, which we are basing our discussion on. Although the constraints specification relies on level of abstraction, the implementation depends on their extent of complexity. The order is from simple to complex. Below is a list of possibilities for implementing the specified constraints applicable to the use-case we are running, see Section 5.3.2. An indication of the constraints inside the web2py Model is made by commenting (uses a comment character in web2py, `#`) at the end of each block of code in Section C.4 of.

1. **Legacies:**
   This category includes set of constraints which have already been taken care of by the Web framework in place. For instance, PRIMARY KEY, FOREIGN KEY, DOMAINS (UNIQUE, NOT NULL, LENGTH) are primitive constraints casually known to web2py. There is no sign of backward incompatibility introduced into these handlers, therefore there is nothing we add here except exploiting them. Therefore, Constraint1 and Constraint2 use existing primitives inside web2py for NOT NULL and UNIQUE constraints, respectively (see Table 5.1). An enforcement of NOT NULL and UNIQUE constraints is made by setting the `notnull` and `unique` parameters of the SQLField to ‘True’. For REFERENTIAL INTEGRITY, one has to point the name of the reference table in place of data type.

2. **Non-legacy - Checks:**
   This is an enforcement of many object-level and table-level constraints using lower-level check constraints (for more explanation, see section 3.6.1). We embraced the `checkerizer` layer inside web2py in order to make an implementation of constraints under this category. It is placed under Section B.1.

   This requires listing of the check expressions (as strings) on a given column or table. Let us see the example down here. The first example defines object-level check constraints. The expression `cidx > 500` verifies cidx records should exceed 500, not included in the constraint specification stage of the design process. The other two expressions in the list, the `ST_isimple(distrib)`, `ST_isvalid(distrib)`, are stored procedures inside PostGIS that return boolean by verifying the simplicity and validity of the indicated geometry column (Constraint3). Plus, the second example defines a table-level check constraint that checks, for every new (distrib) geometry, whether there is a self-overlapping geometry in the same table (again not part of the constraints specification stage).
6.2. The M as a spatial database schema

Basic syntax:
q.define_table([table-name],
    SQLField([column-name], [datatype]),
    ...,
    checker = [comma separated list of check expressions that
        return boolean, and can use stored procedures inside the database])

E.g.:
attribute_checker = ['cidx > 500', 'ST_issimple(distrib)', 'ST_isvalid(distrib)']
table_checker = ['ST_Overlaps(distrib, distrib)']

q.define_table([table-name],
    SQLField('cidx', 'integer'),
    gField('distrib', 'multipolygon'),
    checker = attribute_checker + table_checker)

Look in the example above how the object-level and table-level check expressions are listed, and
assigned to 'checker', an instance of the checkerizer method we introduced into web2py to make
an addition of application-specific check constraints. There is a guarantee that the check constructor distributes a unique name to each check constraints defined, by just adding serials to the
text ‘web2py_checker’ (like this, web2py_checker1, web2py_checker2...). The declaration of check constraints for our use-case can be found in Section C.4.

3. Non-legacy - Triggers:
This class includes advanced checks that involve a bit complex expressions than simpler checks.
One reason of complexity could be the involvement of more than two tables (see section 3.6.1). For
both types of triggers, we considered two options of their implementation.

• Creation of trigger constructor:
For commonly employable constraints that return Boolean, it is believed to be economic
to provide a constructor that urges the developer for some basics. In Constraint4, we said, species distribution geometry should fall within the legal bounding box enveloping Amazonia. This seems a frequently applicable constraint that requires a yes/no answer. Therefore, we produced a constructor for our within example inside web2py that asks the developer for an external table and its column to make the cross-checking with (see B.2 of Appendix B). Its usage looks like below.

Basic syntax:
q.define_table([table-name],
    SQLField([column-name], [datatype]),
    ...
    gField([geometry-column], [geometry-type],
        within=[(external-table, external-column)])

E.g.:
within = ('the_bbox', 'bbox')

q.define_table([table-name],
    SQLField('cidx', 'integer'),
    ...
    gField('distrib', 'multipolygon', within=within))

What is our within trigger constructor doing here? We are pointing the ‘distrib’ column to the external column, ‘the_bbox’ column of table ‘bbox’, a derived table where we stored our
Chapter 6. Implementation: Experiment the re-engineered method

Bounding box geometry. We used a stored GEOS function inside PostGIS, the `envelope`, to derive the bounding box territory from the amazonialegal geometry. The constructor gives to the trigger a unique name, like `web2py_trigger_distribwithinbbox` is the name of our example trigger.

It is also possible to click a second example of this type of constructor. This is a constraint about whether a spatial index should be created or not (discussion about spatial index can be found in Section of Chapter 4). This may not seem a constraint in the narrow sense of the term, but it is an important decision that the Web developer should make at will. For this, we also included an `index constructor` inside the geometry column definer (the last block of code in A.3). A detailed discussion about this geometry column definer exists in Section 4.3.2, while implementing the spatialization of web2py.

By default, the index option of `gField` is set to `True`. Hence, the definition of spatial index does not need special information in our spatialized version of web2py. The user needs to set the index option to `False` if creation of spatial index is thought to be unimportant.

```python
q.define_table([table-name],
               SQLField('cidx', 'integer'),
               ...
               gField('distrib', 'multipolygon', index=False))
```

- **Server-side definition of trigger functions:**
  Definition of trigger functions of this category are similar to the Server-side functions discussed in the next item (Callable functions). Both can be treated in the same manner, except that the return value of the trigger functions is only and only trigger.

4. **Non-legacy - Callable functions:**
   Our new architecture necessitates calling of more than few trigger functions and callable functions. The advantage is momentous. It does not limit the Web developer to few and usually scarce pre-defined indoor functional units (refer Item 3 of Section 3.6.1). However, we don’t also have to forget the computational performance problem that might arise. We have already addressed the issue of performance in Section 8.3.4.

   This category of constraint enforcement, callable functions, encompasses an even more complex set of higher-level constraints that are too complicated to enforce with the help of one of the above ways. Our plan is to make sure that only their signature (function name and list of arguments) is known to the Web. The Web application calls only this signature and fetches what the functions return. The use of Callable functions is the most important choice not only because it takes care of high-level constraints but also it can nest plenty of constraints inside it, including the simple ones enumerated above under checks and triggers. The seven database functions listed under Section 5.2 of the design process are good examples of this category. Their role here is two fold. They can dictate the application logic (as argued in Section 4.3.3), and hold as much and complex constraints as possible inside them. Therefore, there is a need to define one version of callable function that holds inside it as much constraints as possible.

   The creation of highly complicated functions is not the idea of this MSc project. The basic concept and syntax, however, has been introduced under Section 3.6.1. The reader (or the Web programmer) can find a complete instruction about creating functions in the manual of the chosen database platform.

   There are many server-side interfaces utilisable to interact with databases, including to write, test, and debug/troubleshoot database programs. For our prototype, pgAdmin3 was the interface to interact with PostgreSQL on the server-side. Required trigger and callable database functions for our prototype were written in PL/Pgsql language. The callable functions can be found in Section B.3. They are believed not only to handle user-specified constraints but also to enhance the C (of MVC) as discoursed in Section 4.3.3.
6.3. The V as a map interface

6.2.2 Populating the tables

Insertion of data to the non-spatial tables is undemanding, as web2py already handles it using its insert method. But, inserting data into a geometric table calls a totally different approach. One reason of the difference could be the nature of data which is different from the alphanumeric data we are accustomed to. Likewise, the main source of geographic data more often is ESRI shapefile than ordinary forms and text files.

The immediate way to load shapefiles into PostGIS is by passing the command line arguments to the shp2pgsql.exe and psql.exe tools that accompany PostGIS (for detail, see PostgreSQL documentation [38]). The first one creates an intermediary text file of ‘CREATE TABLE’ and ‘INSERT INTO’ statements, SQL grammars to create table and insert data into it, respectively. The second tool reads this text file and loads it into the database.

The application prototype we developed welcomes two possible sources of geometric data as long as they do not violate the constraint rules. First, the command line arguments to sh2pgsql.exe and psql.exe were embedded into a controller action that inputs shapefile from users and appends it into the spatial tables (find the discussion in section 6.4 of this chapter). Secondly, the OpenLayers library has basic functionality that enables users play with geographic data (INSERT, DELETE, UPDATE, VIEW and QUERY) using a map interface that allows basic interactivity such as zooming and panning. Its implementation could be found under Section 6.3 below.

6.3 The V as a map interface

Our application prototype is not navigable; it calculatedly contains only a single but all-inclusive Web page. The view file that contains all the necessary basic HTML and JavaScript functions is called index, show in Section D.1 of Appendix D. It is a structured HTML file ingrains the appearance of the web page shown in Figure 7.1.

Fortunately, web2py's view interprets pure HTML language. In addition to regular HTML tags, python objects returned by functions inside the controller piece can be also embedded by covering them within double curly bracket (like this, {{}}). Inclusion of variables that return python objects instructs the web2py view to generate equivalent HTML statement. What we see on the Web page is this generated HTML, in place of the python objects we put during development. For instance, in the example below, {{=p1}} in V results the following line of paragraph (<p>) tag of HTML, with input (<INPUT>) tag inside it. Its automatic production is at run-time by web2py’s HTML generator. For a deeper look at this discourse, consult the web2py manual (here, [73]).

Generator in controller:

```python
p1 = P('Identifier:', INPUT(_id='cidx', _value='None', _title='species identifier'))
```

```html
{{=p1}} in V:
<p>Identifier:<input id="cidx" title="species identifier" type="text" value="None" /></p>
```

6.3.1 Setting-up spatial servers

Mapserver, Geoserver and, most recently, TinyOWS servers are example standard spatial Web servers that are available in the open source market. Geoserver is a development on top of Apache Tomcat server and the other two are on Apache server. They all are capable of fetching geometric data from the database and flaunting it on the Web. Plus, three of them are capable of importing OpenLayers library for establishment of spatial Web services that we covered in Section 4.2.2, habitually, by fusing it inside <SCRIPT> tag of the HTML (see Section 6.3.2). Only the last two are usable to develop Transactional WFS (WFS-T) application, by transferring data in GML format. Our prototype development utilizes Mapserver and TinyOWS servers for fetching data and viewing them on the Web application only. The reason for this follows.
Chapter 6. Implementation: Experiment the re-engineered method

Usage of these spatial Web servers depends on the three-tier client-server architecture discoursed in Section 1.1.3 of the introductory Chapter. Their role is being a middleware, to process requests coming from the client and responses coming from the database server. Their architecture dictates the receiving and sending of GML based requests and responses. The transaction depends on standard protocol. And, once the request is sent, it is impossible for us to make it callable by our callable functions. However, to guarantee the inclusion of all needed consistency rules, our proposed architectural re-design (see section 3.7) calls for a comprehensive check-up of the inconsistency on the server-side before storage, which is why we needed human-readable and machine readable version of geometries instead, the 'WKT'.

Above all, INSERT/UPDATE requests we want to send by wrapping and firing callable functions (which necessitate the usage of 'WKT' format) are not readable by any of these servers. But, the Web server being used by the Web framework can help us realize such new design of architecture. In our case, Apache is doing this in our case. OpenLayers functionality to serialize geometries was used to hand their readable version over to the callable functions as arguments. The collection of non-geometric attributes uses the universally applied insert forms, before we pass them as arguments into callable functions.

Instructions on how to configure a Web server for web2py can be obtained in the web2py manual (here, [73]). All-inclusive detail about the ‘how-to’ of setting-up Mapserver and TinyOWS can be obtained by visiting their homes at [67] and [98], respectively.

6.3.2 Utilizing OpenLayers API

A closer look at the V of spatial applications hardly bears a difference from the HTML we use in ordinary applications. If there is something which needs special treatment, it is the importation of important functionality from Open Mapping modules such as the OpenLayers API. There are some alternative sources where we can import the OpenLayers module. For instance, the following line imports its sandbox version and makes it ready for JavaScript Scripts that follow it.

```html
<script src="http://dev.openlayers.org/sandbox/adube/openlayers/lib/OpenLayers.js"></script>
</script>
```

The JavaScript function that initializes the map interface on-loading is conventionally named as ‘init()’. The important sections that constitute this function appear below. But before that, its arrangement within the HTML structure should be explained. Once it has been defined inside `<SCRIPT>` tag of the HTML, the rule-of-the-thumb is that we must call it from the `<BODY>` tag of the HTML to allow loading during the initialization of the Web page, as follows.

```html
<body onload="init()">
[body here]
</body>
```

The basic compartments that a fully functional JavaScript map should include are discussed below.

**Base map**

Development of map interface for a standard spatial Web services, such as WFS-T, requires definition of layer as a container of map properties such as projection, resolution, maximum extent, controls etc. (see Section D.2). The map is called mapia in our case. Late addition of controls and properties to it could be made by the calling the following OpenLayers method.

```javascript
mapia.addControl(control)
```

Usually, the first map to set is a base layer beneath all other map layers, which normally takes the name Base Map. Among all most commonly used online maps usable with OpenLayers API, Metacarta
6.3. The V as a map interface

(from [53]) and Google (Google Map, Google satellite and Google Hybrid from [55]) are the four optional bases we chose. Our default base map, that appears first, is Google Map. The lines of code that enables us perform this are shown in Section D.3. Subsequent to the definition of each layer, the following line adds them into the map container.

```
mapia.addLayers([gooSat, gooMap, gooHybrid, basia]);
```

Please be informed that the use of Google maps like above requires an importation of the Google Maps API from any online source, like what the following HTML line is doing it from [55]. For distributed applications, it requires a unique API key. But, for local development purposes, the key seen shown in the following example suffices.

```
<script src="http://maps.google.com/maps?file=api&v=2.47&key=ABQIAAAAAnfs7bkKE82qgGb3Zc2YyS-oBT2yXp_ZAY8.ufC3CFXhHIE1NvwxSySz_REpFq-4WZA270wgbtyR3VcA" type="text/javascript"></script>
```

WFS Vector Layer

Conventionally, WFS transactions are easier to take place if they are in vector format. OpenLayers own “OpenLayers.Layer.Vector” as a vector layer constructor. It helps us define a vector layers and add it into our map defined, as we added the base Map layers in 6.3.2. For more about the coding, see Section D.4). Basically, it needs a feeding of the following arguments:

- **Layer expression** - describes the first string argument that labels the layer added (such as “Amazonia legal”, “Species distribution”, “Interfluves”...)
- **styleMap** (OpenLayers.StyleMap) - describes the default and on-selection style of the layer, which should be declared first (see the portion of the code marked aby //STYLE FIRST in D.4)
- **strategies** (OpenLayers.Strategy) - describes the nature of requests on feature such as:
  - OpenLayers.Strategy.Save – a SAVE strategy which asks OpenLayers to save features on insert/update
  - OpenLayers.Strategy.Fixed – a FIXED strategy that requests features only once and never requests another new data
  - OpenLayers.Strategy.BBOX – a BOUNDINGBOX strategy that request feature data within a bounding box
  - OpenLayers.Strategy.Clustering – a CLUSTERING strategy that requests clustered features, considering the maximum pixel distance between features
- **projection** (OpenLayers.Projection) - specifies what map projection to use (compatible to the spatial reference ID “EPSG:4326” in our case, OpenLayers.Projection(‘EPSG:4326’)),
- **filter** (OpenLayers.Filter) - sorting out the features based on some criteria like:
  - SPATIAL (OpenLayers.Filter.Spatial) – filters based on some spatial criteria, for instance those that fall within a given range of spatial bounding box (OpenLayers.Filter.Spatial.BBOX)
  - COMPARISON (OpenLayers.Filter.Comparison) - filters based on some specific range of values/attributes, such as:
    * greater than (OpenLayers.Filter.Comparison.GREATER_THAN, ‘>’),
    * less than (OpenLayers.Filter.Comparison.LESS_THAN, ‘<’)
    * equal to (OpenLayers.Filter.Comparison.EQUAL_TO, ‘==’)
Chapter 6. Implementation: Experiment the re-engineered method

- FeatureID (OpenLayers.Filter.FeatureId) - singles out a specific feature by using just its unique id
- LOGICAL (OpenLayers.Filter.Logical) - combination of any two or more of the above using operators such as:
  * AND (OpenLayers.Filter.Logical.AND, ‘&&’),
  * OR (OpenLayers.Filter.Logical.OR ‘||’),
  * NOT (OpenLayers.Filter.Logical.NOT, ‘!’)

Note: Our prototype required a COMPARISON filter of species distribution based on species id, since unfiltered species distribution by its very nature results in extremely overlapping geometries to the extent that it is impossible to separate the ones underneath. (see the block marked //FILTER in section D.4)

- protocol - answers the question which OGC®-compliant protocol (which is WFS in our case, OpenLayers.Protocol.WFS) (as exposed in Section 4.2), with following details:
  - readFormat - the compatible vector format for reading data (OpenLayers.Format.GML())
  - outputFormat - the compatible vector format for display (“GML2”)
  - version - to create a versioned protocol of WFS (“1.0.0”)
  - url - the URL to send the requests to (“/cgi-bin/tinyows.exe”)
  - featureType - unique name of the feature layer (e.g., “distribution”)
  - srsName - spatial reference system (“EPSG: 4326”)
  - featurePrefix - feature namespace alias (e.g., “dis” for distribution)
  - featureNS - Feature namespace (domain name, e.g. “http://localhost/”)
  - featureName - name of the geometry column of the layer feature (e.g., “the_geom”)
  - schema - a string of DescribeFeatureType request that describes the feature (e.g., “/cgi-bin/tinyows.exe?service=WFS&version=1.0.0&request=DescribeFeatureType&TypeName=amazonialegal”)

Also, OpenLayers supports the serialization of geometries in all basic vector formats such as WKT, GML2 and GML3, GeoJSON, GeoRSS and Google’s KML. By taking advantage of this functionality, the serialized ‘WKT’ version of the geometries are considered appropriate inputs of our callable functions that include application transactions of our WFS-T.

Access to each layer needs a configuration file that defines it and its basic properties (projection, connection to the database, formats etc.). These are a text file named config.map in Mapserver and an xml file named config.xml in TinyOWS (For more, see Section 6.3.1).

Controls and Editing Tools

In addition to the controls added at the initial definition of the map container in Section 6.3.2, our WFS-T application needs some more controlling tools, such as navigating, drawing, modifying, deleting and so on tools (see Section D.5). Fundamentally, these tools are accompanied with functions that trigger on their activation and/or deactivation. The major tools are placed on an editing panel created by calling OpenLayers.Control.Panel (see the portion of the code in Section D.5 marked //PANEL). We can add controls to the map container and activate them by the following lines of code, paneel being a panel object.

```javascript
paneel.addControl(control)
control.activate()
```

6.3. The V as a map interface

**Navigation history (OpenLayers.Control.NavigationHistory)** – It creates an object \( \text{navtry} \) that keeps records of navigational history forwards (\( \text{navtry.next} \)) and backwards (\( \text{navtry.previous} \)) (see \//NAVIGATION HISTORY mark in Section D.5).

**Zooming** – OpenLayers also provide the zooming interactivity in terms of “Zoom in” (OpenLayers.Control.ZoomBox), “Zoom out” (OpenLayers.Control.ZoomBox, with the option ‘out’ set to true) and “Zoom to maximum extent” (OpenLayers.Control.ZoomToMaxExtent). They are referred zoomIN, zoomOUT and zoom2max in our prototype (see \//ZOOMING mark in Section D.5).

**Selection (OpenLayers.Control.SelectFeature)** – As its name indicates, it selects the feature on click. It is named selector in our case (see \//SELECTION mark in Section D.5)

**Highlight (OpenLayers.Control.HighlightFeature)** – gives the possibility to highlight feature on mouse click or hover. The additional character of feature highlighting is the popping up of attributes of the highlighted feature on a styled pop-up form. This is possible by setting the displayPopup argument to true and defining its pop-up size \([\text{popupSize}]\) and style (see the highlighture object shown in the \//HIGHLIGHT portion of Section D.5).

**Draw and INSERT geometry (OpenLayers.Control.DrawFeature)** – Offers a handler to draw a Point feature (OpenLayers.Handler.Point), Line feature (OpenLayers.Handler.Path) and Polygon feature (OpenLayers.Handler.Polygon). Species distribution is polygonal, and the last drawing tool was used (we called it drawor, see \//INSERT block in Section D.5). It basically requires specification of the name of feature layer to-be stored. It could possibly be accompanied by events that fire before/after it is activated. For example, the following line registers the \( \text{featAdded} \) event (locate the \( \text{featAdded} \) function in Section E.1) to fire once a feature is added (condition = “featureadded”). Meaning, \( \text{featAdded} \) function triggers immediately after the feature drawing ends to activate the feature editor tool which, among other things, pops up the ‘attribute form’ for inserting attributes of the feature being selected (see also discussion on ‘modify’ tool below).

\[
\text{drawor.events.register('(\'featureadded\', \'', \text{featAdded});}
\]

**Modify and UPDATE geometry (OpenLayers.Control.ModifyFeature)** – This is the most important functionality needed for data-intensive WFS-T applications. It enables us toggle geometries for not only an update but also an insert (see \( \text{feaditor} \) definition shown in the \//UPDATE portion of Section D.5). Like the “draw” tool, it can be preceded or followed by event triggers, such as \( \text{onEditorStart} \) and \( \text{onEdit} \) that launch before feature modification and during feature modification, respectively (see Section E.1).

\[
\text{wfsTure.events.register("beforefeaturemodified","", onEditorStart);} \\text{wfsTure.events.register("featuremodified","", onEdit);} \]

The most basic two actions performed by \( \text{onEditorStart} \) JavaScript (JS) function are adding a form pop-up for inputting attributes of the feature on modification (\( \text{addFormPopup} \)) \( \text{QoQ} \) being the feature selected) and parsing feature attributes to fill fields of attribute form that pops-up by fetching values from their corresponding fields stored from the database, if any. For parsing, we should call another function \( \text{parsefeatAttr2Form} \) \( \text{QoQ, featAttr} \) \( \text{QoQ} \) being the feature selected and \( \text{featAttr} \) being the attributes of the same feature as a response of a HTTP GET request shown below (see also, the last block of codes in Section D.5). Consequently, \( \text{setAttributes}() \) is another JS code that sets the response received. During modification, \( \text{onEdit} \) function is called to identify the feature on modification for debugging purpose (OpenLayers.Console.log).
Chapter 6. Implementation: Experiment the re-engineered method

In addition, events can be made to happen while an editing tool is active or inactive. A very good example of this situation are the onEditorActivated and onEditorDeactivated (locate them in Section E.1). These are the two executables on activation and deactivation of the modifying tool, to automatically activate and deactivate the feature highlighter tool (highlighture), respectively.

```javascript
feaditor.events.register('activate', '', onEditorActivated);
feaditor.events.register('deactivate', '', onEditorDeactivated);
```

**Delete (OpenLayers.Control.DeleteFeature)** — Delete tool is one of the transactional tools OpenLayers offers (see the //DELETE mark in D.5). It gives an interface-based opportunity to remove geometries once and for all. Our prototype also defined two before-event functions, one that invokes before the feature deletion request performs (normally, to send a message to the user for confirmation of the action, beforeDel in our case) and second that invokes before removal of the deletable feature (while deletion has already been confirmed to happen by the user, onBeforeRemoved).

```javascript
delor.events.register('beforefeaturesdeleted', '', beforeDel);
wfsTure.events.register('beforefeatureremoved', this, onBeforeRemoved);
```

To know which features are being deleted before action, the onDelActivated function activates the feature highlighter tool, and the onDelDeactivated undoes the the highlighter activation.

```javascript
delor.events.register('activate', '', onDelActivated);
delor.events.register('deactivate', '', onDelDeactivated);
```

### 6.3.3 CSS

CSS is a standard and simplified document attached to HTML documents to characterize the appearance and style of fonts, colors and spacing of Web pages. It define set of rules to be embedded inside HTML tags and dictate the presentation and layout of Web pages. Our WFS-T user-interface is aware of the style file that has been imported from the draft style sheet available in OpenLayers page. One can download it from [3] and make a manual edition as it fits his/her purpose.

**Flash**

To make a Web application feel lively, web2py provides a flashing layout to send response message (usually error messages) to the Web client. This resource of web2py were highly exploited to help in the development of our prototype for the proper communication between the server and client, to catch error messages and send them to the user.

### 6.4 The C as a mediator

The controller as a component that controls the application logic hosts the definition and return of the following basic actions, described below (see the functions in Appendix F):

- **Index** - is the function that returns some basic HTMLs while initially loading the application page, generated by web2py’s HTML generator (html.py). The ‘INPUT’, ‘P’, ‘FORM’ and ‘OPTION’ generate the `<INPUT>`, `<P>`, `<FORM>` and `<OPTION>` HTML elements respectively. And in the View part, their return value are used inside a double curly bracket (as `{{return value}}`) to let web2py differentiate pure HTML codes from python ones (see Section 6.3).
• **Update** - is the most influential function in the transactional application we developed; fires when users push the button ‘UPDATE GEOMETRY’. It includes three main constituent parts, the ‘insert’, the ‘update’ and ‘delete’, in which only one executes at a time depending on the state of database transaction returned by the OpenLayers.State functionality in OpenLayers. They fire the `add_distrib`, `update_distrib` and `delete_distrib` callable functions, respectively. Web2py gets the state information together with the basic resources about the on-modification feature, such as unique-identifier, the geometry and its other attributes, from the hidden form that collects them after serialization of the geometry (see `plzSerialize()` function in Appendix E). The geometric data are inputs to the three equivalent callable functions. And, the non-geometric data are inputs to `add_species` function described in 5.2.2 of Chapter 5. If transactional or other errors occur, a message that informs the user that an error has occurred is sent, by making use of the flash design of web2py.

• **restrict** - the action that restricts species distribution geometry into interfluvial boundaries, specifies by selecting them. It expects the collection of interfluvial geometries in serialized ‘WKT’ format and gives it to the `restrict_distrib` as an argument. By firing the callable functions, it sends a success or failure report to the Web application.

• **addFam** - this is an action that gets inputs for the `add_family` function from the family form. Once the callable function is fired, it sends a success or failure report to the Web application.

• **addGen** - this is also an action similar to addFam above, except that it collects inputs from genus form for the `add_genus` function.

• **shp2pg** - this is another action that gives an option to geometric insertion, in addition to the drawing and inserting tool OpenLayers provided us. It asks users to browse for a shapefile and it passes it together with the optional commands to `shp2pgsql.exe` compiler. For the full description of the options, the user is directed to the PostgrGIS documentation in [58]. But below, the flags used for our prototype will be described:

  – Path to shapefile – points to where the shapefile is located in the local drive
  – -g – naming of the geometry column
  – -s – to-be used SRID (e.g., 4326)
  – -a – (append), tells the compiler to append the shapefile into the existing table with the given name and geometry column. Mutually exclusive options are:
    * -d – Drops the existing table with the same name, recreates it to populate it with the provided shapefile
    * -c – Creates new table to populate it with the provided shapefile
    * -p – (Prepare mode), just creates the table, no further action.
  – path to SQL file - Points to where to save the automatically generated SQL file

The automatically generated output, the intermediary SQL file, needs passing as input to `psql.exe` compiler to finally load the shapefile onto the database. `psql.exe` also requires at least the following command flags, basically for database connectivity:

• Path to SQL file - Points to where the automatically generated SQL file is saved
• -h – (host), the host server where the database is running
• -d – (database), the name of database where to execute the SQL command
• -U – (USER), username to access the database. A pop-up window appears for password authentication if the provided user name is not valid and trusted
6.5 Conclusion

All our previous chapters were jumping platforms for the current chapter. What were discussed in the last few chapters became practical in this chapter. It also followed the lines of demarcation of MVC.

The first thing to do was to implement the M (of MVC) as spatial database schema. So, it was a physical realization of the Content model designed in Section 5.3 of the design process in Chapter 5, by finally making it particular to PostgreSQL and then web2py. The main part of the work follows the steps we took to enrich web2py to make it follow the freshly re-engineered client-server architecture in Section 3.7. This way, it was found realizable to make an enforcement of constraints at different levels of complexity, going from simple to relatively complex ones.

An implementation of the V (of MVC) in this chapter exploited the OpenLayers API. Basic functional units of OpenLayers helpful for the development of our example WFS-T prototype were used. We required some JavaScript coding to make the application more responsive.

Also, the C (of MVC) interceded between the M and V to define the flow of the application logic. The basic components of the functionality of our WFS-T prototype got life with the help of python-coded actions in the C of web2py. It introduced new concept of firing callable database functions for their server-side execution, instead of their commonly applied execution on the client-side. At least, this was required for the realization of a fully functional and self-sufficient spatial database than the one that used to be used only for storage.
6.5. Conclusion
Chapter 7

Experimentation: Experiment the implementation

7.1 Introduction

The evaluation of the developed system was made by raising few questions that assure whether it is serving its intended purpose or not. This is to mean, if only valid data are admitted into the database and any sort of action that is suspected to contaminate the database content is totally protected and handled in a proper manner. This chapter gives a brief demonstration of the WFS-T application by showing to what extent spatial database consistency is taken care of. First, the appearance of the application page at opening was briefed in Section 7.2. Following is a guidance on how to use each control tools included in the application, in Section 7.3. Section 7.4 illustrates the interactive usage of the WFS-T prototype to directly make transactional communication with the spatial database at the back-end.

It is yet known that the most important intention of our project was not to create a WFS-T application, rather to make it a consistency-preserving one. In order to make this evaluation easier, a question were raised, to see what would happen if the Web application is forced to perform an invalid action. One way of doing this is testing it by providing invalid inputs that are expected to violate the defined constraints, such as invalid species distribution geometry that falls outside the allowed bounding box (Constraint4, Section 5.2.3). If such actions are trapped well enough and their transaction not allowed to be committed, there is a guarantee for us to say our ‘WFS-T’ prototype developed using web2py is ‘CONSISTENT’ enough. Hence, an example was demonstrated in Section 7.5.

7.2 The facade

The index page of the our web2py-powered CONSISTENT WFS-T prototype is composed of the following HTML components (see Figure 7.1, and see Section 7.3 to see how to make use of each page component):

- **TITLE** – title of the page (which is ‘CONSISTENT web2py-powered WFS-T’)
- **Map window** – with the following facets inside it
  - Default base map – which is the Google Satellite Map,
  - Vector layers – Overlay of interfluves, Amazonia Legal boundary and Species distribution (which requires filtering, see below)
  - Pick-list by species id – to filter and display only a single species distribution at a time (see Section 7.3.2)
  - Layer switcher – To put on/off base map and vector layer options (see Section 7.3.1)
  - Editing tools – The OpenLayers tools that trigger different actions, placed inside a controls panel
7.3 Playing with the controls

— Zooming slider and navigation arrows – OpenLayers tools for zooming in/out by sliding and navigating by clicking directional arrows
— Overview window – to navigate around by approximating the relative position on a small rectangle
— Important BUTTONS:
  * UPDATE GEOMETRY – that fires the ‘Update’ controller function (See Section 7.4.2, and for more about the functions see Section 6.4)
  * RESTRICT DISTRIBUTION – that fires the ‘restrict’ controller function (See Section 7.4.3, , and for more about the functions see Section 6.4)

• Shapefile browser/uploader – An upload form that enables users to browse for ESRI shapefile about species distribution and append it onto the distribution table of the spatial database

Once the legacy species distribution, amzonialegal and interfluves shapefiles (the optimal bounding box included) are uploaded onto the database back-end, the page looks like as it appears in 7.1.

Figure 7.1: The facade of our web2py-powered CONSISTENT WFS-T prototype

7.3 Playing with the controls

The main map controls embedded in our WFS-T prototype to grant it flexible playability are discussed as follows.

7.3.1 Layer Switcher

The Web application encompasses seven layers of map, with four of them serving as base layers and the remaining as vector overlays (see \[ \] sign at top-right corner of the map window shown in Figure 7.1). Clicking this icon stretches the contracted small blue window that holds alternative overlays and layers. Changing the default base map, i.e., Google Satellite map, is possible by checking the radio button in front of each base map layer. Checking/unchecking boxes in front of vector overlays switches it on/off. See the small blue window in Figure 7.2.
7.3.2 Filtering species distribution geometry

This is not a control in its narrowest sense. However, it is equally important tool in our WFS-T prototype. Unsurprisingly, bird species have many common characteristics and their geographic habitat distributions form extremely non-separable overlapping polygons. Hence, we applied the filtering mechanisms of OpenLayers to display a sole distribution data at a singular time. The criteria for filtering could be specified by picking a species id from the drop-down box at the top-right of the map window. At first, ‘none’ is selected and none of them are displayed like in Figure 7.1. See the example in Figure 7.3 that shows the distribution geometry of species number 967. For contrast, we also presented the jam-packed unfiltered species distribution layer in Figure 7.4.

7.3.3 Zooming

In/decreasing the focus of the map viewer, one can play with the Zooming tools, which come in different options listed below:

- Zoom to maximum extent (uses the icon inside the control panel) – Stretches the base map in to its maximum view.
- Zoom by scrolling middle mouth-button – Scrolling up/down the middle mouth ball zooms in/out the viewer into the next upper/lower level of map scale
- Zoom by box – First one should activate the zoom in/out tools from the control panel (use icons for zoom-in and for zoom-out). Drawing a rectangle by mouth dragging at the target portion of the map changes the zoom extent accordingly (see Figure 7.5); active icons will directly change to . Simple double-click without mouth dragging could also do the same.
- Zoom by slider – Panning the vertical zoom slider up/down changes the zoom extent in/out accordingly

7.3.4 Map scale and mouth locator

The other non-trivial additional map information included in the Web map viewer are the OpenLayers fixed scale line and counter-like mouth position teller. The former feeds information about the average
7.3. Playing with the controls

Figure 7.3: Filtering example: distribution of Species 967

Figure 7.4: Unfiltered version of species distribution layer
scale of the map and the later show the lat/lon position of the place where the mouth pointer is currently hovering. They are positioned at the lower-left and lower-right corners of the map div by default (see Figure 7.6, both encircled by [ ]).

7.3.5 Navigation and history

To navigate around the map window, the following few possibilities can be employed:

- Navigation by panning – The hand-like tool ([], lined first in the control panel) is for a simple drag-mouth panning,

- Directed arrows – the directed navigators [ ] above the zoom-slider move the map viewer to the same direction they are pointing to.
7.4. Data inserts and updates

- Map Overview window – Hidden in the [+] sign at the bottom-right corner right above the mouth position, the red dotted-rectangle inside the small blue map overview window approximates the current position of the map in the view (see Figure 7.7). Moving it changes the relative position of the map accordingly.

- Forward/backward navigation history – The two last arrowed tools at the editing panel keep a record of the navigational moves made. The first recall all previous instances of navigation made and the second on the reverse takes to the next move.

Figure 7.7: OpenLayers control tools: Map Overview window

7.3.6 Feature attribute pop-up

Before ‘DELETE’ or ‘UPDATE’ features, our interface provides information about the current feature on mouth hovering, to supplement the user with extra information to let them know what they are doing before action (like it is shown in figure 7.8).

7.4 Data inserts and updates

Obviously, the the main four database operations to-be performed by our WFS-T application are inserting new species, distribution, family and genus data and updating existing ones. Deleting geometry data and restricting the distribution geometry into a list of interfluvial catchment area are the other ones (see Section 5.2.2). By following the newly re-designed MVC set-up (see Section 3.7), it was intended to be consistent, in the sense of the term as introduced in Section 3.2. The following section raises and answers a bold question related to the state of consistency our prototype enforces. Before that, let us see how to make use of the insert/update and delete tools.

7.4.1 INSERT species distribution data

To insert a species distribution geometry by drawing, the user has to activate the OpenLayers draw polygon [ ] tool (placed 6th in the top-left control panel) and the icon changes to [ ]. Then, drag the mouth over the map area where we want to add geometry. Each click adds a geometry node and double-click ends the drawing by finally triggering the pop-up attribute-form that inputs the basic properties of the species whose geometry is in case (see Figure 7.9). Each field defaulted the value ‘None’. The ‘SERIALIZE’ button in turn triggers the plzSerialize() (see Appendix E) action that serializes the geometry at-hand into a ‘WKT’ vector format and makes it ready for insert, together with species attributes.

The ‘UPDATE GEOMETRY’ button at the bottom-left corner of the map takes care of the rest, to send the ‘save changes’ command to the database, by firing update() controller function (see Section 6.4).
Chapter 7. Experimentation: Experiment the implementation

Figure 7.8: OpenLayers control tools: feature info

Figure 7.9: INSERTION of species distribution data
which first collects and arranges the serialized geometry and its attributes. It then sends the ‘INSERT’ command into the database once the transaction state is found to be so.

Up on successful completion of the spatial database transaction, the database comes with a success message. If the spatial database transaction fails to commit successfully, failure message is sent instead. Such response messages are made using the powerful flashing functionality of web2py.

7.4.2 UPDATE species distribution data

Updating existing geometries is not much different. By first filtering the to-be updated species distribution geometry (picked from the top-right filtering picklist by id), selection of the feature using \[\text{button}\] goes next (placed 7th in the control panel) and the icon changes to \[\text{icon}\]. A click on the to-be-updated geometry toggles its topological structure and triggers the attribute-form. This is the same as it looks in Figure 7.9 but each fields filled with corresponding values stored in the database, if any (see Figure 7.10). Dragging the nodes of the geometry, with left-mouth button stayed clicked, changes it as required. ‘SERIALIZE’ and ‘UPDATE GEOMETRY’ functions and goes the same way as it is explained above for geometric insert.

![Figure 7.10: UPDATING of species distribution data](image)

7.4.3 RESTRICT species distribution geometry to given list of interfluves

The discussion about our use-case in the design process (Section 5.2.2) revealed that the interfluves boundaries within the defined bounding box dictate the distribution of species. It is, therefore, necessary to put in place another version of geometric update that plots species localities based on an indication of their known possible interfluvial locations.

Which species is required to be restricted within specified list of interfluves? This being the first question in this process, it can be answered by picking a species of our interest from the right-top pick-down list, by id. The next step, indication of the possible interfluvial geometries for the current species, is made by feature ‘SELECTION’. After an activation of the ‘SELECT’ tool (the third icon \[\text{icon}\] in the control panel), a click with ‘SHIFT’ key remained pressed multiplies the selection, whereas ‘CTRL’ key undoes the ‘SELECTION’) and clicking on the interfluvial polygons where the currently selected species is expected to reside prepares the callable arguments for restriction (see Figure 7.11, highlighted are selected interfluvial catchment areas to be restricted to the picked species).
Pushing the ‘RESTRICT GEOMETRY’ button fires the `restrict()` controller action that sends an update request of the species distribution (based on the interfluves restriction) to the database. It wraps the species feature identifier and a list of ‘WKT’ geometries of the specified interfluves. A ‘UNION’ (defined by the ‘ST_UNION’ function of GEOS module inside PostGIS) of these geometries changes the current species distribution, if and only if it accords with the pre-described constraint rules.

![Figure 7.11: RESTRICT species distribution geometry to given list of interfluves](image)

### 7.4.4 DELETE species distribution data

To remove species distribution geometry and its attributes ones for all is another transactional function performed by our WFS-T prototype. The button in the interface that enables us make data deletion is represented by ![icon] icon under the editing panel. Already stored geometries would not be immediately disappear while selected by the ‘DELETE’ button, rather the order waits the pressing of ‘DELETE’ key on the keyboard to alert a before-deletion confirmation message. An approved ‘DELETE’ action followed by push of the ‘UPDATE GEOMETRY’ button commits successfully if and only if it does not violate any constraint rule defined, especially if REFERENTIAL INTEGRITY constraint (Constraint5 5.3.1).

### 7.4.5 INSERT family data

To add family data, into family table of the database, a form that collects the attributes should be first made ready. For this, clicking the insert family icon in the control panel ![icon] pops up the family form (see Figure 7.13). Pushing the ‘Save Changes’ button triggers the `addFam()` controller action (see Section 6.4, which in turn receives the input attributes to feed it to `add_family` callable database function as arguments and fire it.

### 7.4.6 INSERT genus data

Steps of adding genus data into genus table of the database, is similar to adding family data uncovered in Section 7.4.5 above. Activation of the insert genus icon in the editing panel ![icon] pops up the genus form (see Figure 7.14). Once we fill the form inputs, a push to the ‘Save Changes’ button triggers `addGen` controller action (see Section 6.4). It takes arranges the received form inputs as arguments to the `add_genus` callable database function (discussed in Section 5.2.2 of Chapter 5).
7.4. Data inserts and updates

Figure 7.12: DELETION of species distribution data

Figure 7.13: INSERT of family data
7.5 Preserving spatial database consistency

To prove how consistent our web2py-powered WFS-T prototype is, an important question that requires an valid answer were raised. Our example in Figure 7.15 and Figure 7.16 exhibits what is going to happen if an attempt to violate the boldest (and most spatial) example constraint, ‘INSERTING/UPDATING’ of species distribution data which falls outside legal bounding box (Constraint4, see Section 5.2.3). The answer is, the database will such violations of constraints by tracking them back to where they come from, the Web application. This is the result of the new architectural arrangement we introduced in Section 3.7.

7.6 Conclusion

The consistent WFS-T prototype developed in Chapter 6 get experimented in this chapter. It was proved that our deliberate trail to invalidate one of the rules defined for our use-case Section 5.2.3 and Section 5.3.1 were well taken care of by the database. At this level, we dare to conclude that the our proposed architecture by considering options of constraints handling from the database world were successful in the sense of trapping invalid actions that invalidate the spatial database consistency.

In this chapter, by first introducing our WFS-T prototype in Section 7.2, a tour were made in Section 7.3 to demonstrate the functional tools and their playability, and in Section 7.4 to their use in order to trigger the data-based functions devised during the RE phase of the design process. An example that shows when invalid geometry insert got controlled by the database following the new proposed architecture we presented in Chapter 7.5. With this, we ended up the Chapter.
7.6. Conclusion

Figure 7.15: INSERT of invalid geometry data - outside legal bounding box

Figure 7.16: Response given from the database as a result of an attempt to insert invalid geometry
Chapter 8

Summary, conclusions, recommendations and impact

At the outset, section 8.1 presents major findings and achievements this thesis adds to the science value, in the form of summary. It scans and analyzes the major dealings of this MSc research project in a light-weight and birds-eye view. Section 8.2 takes us to set of conclusions that show there is no research questions and/or objectives left completely unanswered. However, some shortcomings of the study are pointed out and put as recommendations for further advanced research in section 8.3. The impact we know our particular discourse on web2py might have on other, but MVC-based, Web frameworks continues in section 8.4. With this, we wind up the main subject under discussion.

8.1 Summary of the study

The main objective we set out to accomplish during the MSc research period was to forward an alternative solution to Web frameworks modeling that enables the comprehensive management of not only pre-defined but also user-defined application-specific spatial consistency rules. This was motivated by the limitedness, both in quality and quantity, of the MVC-based Web frameworks set-up that facilitates the control of only pre-defined functional constraints, delineated for spatial databases.

We first screened top-six python-powered Web frameworks, which depend on analogous paradigm and work with PostGIS-extended PostgreSQL, the spatial database platform of our choice. By conducting a comparative analysis to them, web2py was found to be satisfying some yardsticks required for this MSc project. Therefore, all inferences and tests to MVC-based Web frameworks throughout the thesis have been made particularly via web2py.

The research work began by inquiring ‘why are databases utilized only for storage purpose in the Client-Server Architecture of MVC-led Web applications development process, while still they can offer a lot more benefits including the comprehensive consistency control?’. This is the major drawback our research project deemed to seek a potential solution for. The optimal research approach we followed was to first search for model-led Web engineering approach (Chapter 2) and practical way-outs the database technology could offer us (Chapter 3). Industry-based modeling principles of WDA were scanned, to finally pick UWE that bases its theory on established standards, especially the UML. The other strength of UWE were found to be its overall structure that separates autonomous concerns, among which the content piece is the one that deals with the practices that can be directly and entirely inherited from database technology. This is where we put our focus to later outline up-to-the-minute methods to bring them into an integrated design frame, tested and implemented in Chapter 6. Hence, we defined a new architecture that allow Web designers identify data constraints at different abstraction layers (object-, table- and database-level) and enforce them at different degree of complexity (check, triggers and callable functions).

Despite our first choice, web2py was non-spatial in the sense that it does not support the development of any standard spatial Web application at all. This fact refers to not only web2py but also many of the Web frameworks pursuing the same architectural design philosophy. This made the Web-based spatial database consistency dealing anticipated by this MSc project unattainable without making a spatial
annexing to the Web framework of choice, web2py. Hence, it called for another redesign of its internal structure and an implementation following the newly proposed generic design. This is documented in Chapter 4 in such a way that it is directly re-usable for spatial enhancement to other Web frameworks. We created a generic guideline on how to make a spatial extension to MVC-led Web and tested it on web2py (Chapter 4). This way, we successfully developed the first web2py-powered Transactional Web Feature Service (WFS-T) that enables privileged Web users DISPLAY, INSERT, UPDATE, DELETE and QUERY spatial objects stored in spatial Database at the back-end (Chapter 6). It use-cases a working project on Web-based content management of bird species distribution of the Amazonia river, whose design process in Chapter 5 which were made based on the new re-designed architecture in Chapter 3. We make a demonstration of our Web2py-powered WFS-T prototype in Chapter 7, by highlighting its support to spatial database consistency.

8.2 Conclusions of the study

Incidentally, this Master thesis project brought some novel findings and contributed few inputs into the Web frameworks engineering, seen as an even more powerful amalgamation of the three influential technologies in computing world: the Web, the Database and the Network. The principal ones are stratified under the following categories, by raising questions which are related to the research questions and objectives this MSc project strive to answer.

**Was it found possible to base the modeling of the major concerns in WDAs, especially the database compartment, on formal and so-called effective design principles?**

Yes, in deed! Chapter 2 reviews all methods that base themselves on the modern theories of WE. One of the dimensions in Web-database applications was found to deal with from-simple-to-complex design phases to model the so-called ‘levels’ dimension, which lists the central concerns into three: the Content, the Navigation and the Presentation. Therefore, design-driven modeling of major concerns in Web-database application is one of the core principles behind all WE approaches. Web Engineering’s care for the ‘Content’ level is comparatively concurrent to the practices of the database technology. We exploited this underlying concept in our design process, Chapter 5.

Plus, MDA relies on models and their transformations at different phases of the system development, except that we did not define transformation rules with spatial constraints included (see Section 8.3.3). It first requires simplification of system functionality without involving any computational technology, to the extent of using even paper and pencil to make a sketching, and results the the CIM. Such model should be mapped to a more-or-less standardized but irrespective of the Web framework and database platforms of choice. It is at this stage, which is known as the PIM that the design-driven modeling practice of the database technology matches to the MDA-compliant Content modeling of a Web-database application. the PIM can later be mapped to coding primitives, called the PSM, that would be specific and deployable in favorite platforms.

However, Web applications developed with the help of Web frameworks were not found submissive to such WE tradition. Hence, a design frame that fits the MVC architecture into the MDA-compliant WE approach has been proposed. In the middle of this route, we were in favor of UML-based Web Engineering (UWE), since it was recognized to work hand-in-glove with many industry-based standards.

**Was it found possible to take care of beyond-the-subset consistency rules that involve in MDAs built with the help of Web frameworks?**

Yes, indeed! Once we formalized the a MDA-compliant WE method for Web-database application development process using Web frameworks, the idea behind spatial database consistency have been treated in Chapter 3. By first situating a working and functional definition of the term ‘consistency’, the theory of ‘topological model’ as an eminent building block in the study of the internal structure and external relationships of entities in space were covered in some detail. The section that follows tries to furnish literature-based taxonomy of constraints, they being rules defined for consistency of spatial databases.

At the cutting edge, that chapter discussed the specification of constraints based on their levels of abstraction and implementation of constraints based on their levels of complexity. This way, the MVC-led
architecture of Web frameworks has been re-engineered to make the most out of the potential resources at the back-end for constraints specification and implementation. Developing a standard spatial Web application that requires high data trafficking to and from the database were essential to examine the effectiveness of this newly proposed architectural solution. That is why we implemented the WFS-T prototype in Chapter 6, on a use-case of a working project about the bird species distribution of the Amazonia river. The prototype was found effective in the scale that it adequately handles all user-specified constraint rules, which we supposedly referred them as beyond-the-subset rules in Section 1.2 of Chapter 1. One of such prominent constraints is the rule that controls species distribution data which does not fall within the legalized bounding box of the Amazonia vicinity. Invalid actions that force the Web application to input such invalid geometry data were proficiently obstructed by our prototype, as we intended it to be.

Was it found possible to still preserve the strengths of the constraint handlers already implemented by the Web frameworks in place?

Fair enough, via making an effort to crack the inadequacy of spatial database consistency handlers inside Web frameworks engineering, we were aware of the fact that there are some strong points already in place. We also determinedly inherited many of these strengths in our implementation, both while spatializing Web2py and developing our WFS-T prototype.

For instance, there are some constraint handlers already implemented and conserved by the existing configuration inside web2py. An example includes the NOT NULL and UNIQUENESS constraints. And, hence their strength was well appreciated and preserved. However, they are not formalized in a way to principally deal with the consistency issue of spatial databases. They are treated as ‘Legacy’ constraints as part of the constraints implementation phase in our projected architectural re-engineering.

Was it found possible to sketch and implement a generic method of spatializing MVC-led non-spatial Web frameworks?

There is no more compelling answer to this question than seeing the first spatial application built using web2py, which was formerly far from the spatial sphere of influence. In Chapter 4, generic enhancements rules of the three MVC components were prepared by utilizing the state-of-the-art methods and open-source tools available in the market. And, the augmentation made to MVC has been valuably tested on web2py. It is this spatialized version of web2py that powered our consistent WFS-T prototype development in Chapter 6.

This might not make web2py a full-fledged world-class spatial framework, but it requires some more primitive amplification which is far beyond the control of this MSc thesis project. However, posterior to this MSc project, there is an arrangement to keep the initiative in motion by the author.

8.3 Recommendations of the study

Standing on the WE achievement this MSc thesis brought, it is perceptibly wise to forward potential areas where further research project could credibly be conducted. This is expected to get our accomplishments much better. The following recommendations are formulated for this MSc research project and they are discussed right below.

- Extend other constraint implementation options
- Utilize OCL-based constraint specification
- Make a full integration of the MDA
- Resolve performance problem
- Realize Spatial enrichments to OCL
- Enhance the current functionality of our WFS-T prototype
- Utilize extended version of UML for MVC modeling
8.3. Recommendations of the study

- Put a powerful wrapper for creation of useful database functions, callable from the Web later
- Comprise exception handlers inside Web frameworks
- Make a full-fledged Web-database application design
- Intensify the current guideline for spatialization of Web frameworks

8.3.1 Extend other constraint implementation options

Apart from the constraint implementation option we chased in our thesis project, we value the actuality that there are many other ways to handle constraints by an autonomous database at the back-side of the client-server architecture. Many of such alternative ways were discussed in the constraints implementation section (Section 3.6). We also would like to declare here that this is not all there is. Extending the range of options for constraints enforcement is expected to boost the effectiveness and self-sufficiency of the spatial database that involves in a given Web-database application.

8.3.2 Utilize OCL-based constraint specification

Among all the constraint implementation methods discussed in Section 3.6 of Chapter 3, the benefit of using the option that advocates the use of OCL was seen in two ways. First, it gives the designer a chance to enforce constraints by first specifying them as part of the data model. Secondly, it is very much compliant to the MDA and the WE method we employed for our project, the UWE (see Section 2.4. It was due to time constraint that we did specify our constraints manually in Section 5.3, instead of employing the standard language for constraint specification, the OCL by integrating them with rules of MDA model transformation rules. Further study should underline this point of fact.

8.3.3 Make a full integration of the MDA

Even though our design frame adopted in Section 2.5 tried to integrate the notion of MDA into the MVC triad, it is an unfinished task because there is no step taken to fully and automatically map models produced at each level of concern into the next level. Many research work such as [65] and [99] not only defined model transformation rules and generic guideline but also treated automatic OCL-based specification of spatial constraints as part of the MDA-compliant modeling and model transformation. It was wise to inherit such approach to our design frame. That would help for a successful implementation of MDA based code generation for WDA development. This is believed to sharply shift the culture of Web frameworks. Hence, a recommendation is formulated for a further research to intensify the MDA practice.

8.3.4 Resolve performance problem

In distributed applications, with WDA as a best possible example, it is good practice to think of the performance and computational cost issue that involve in exceedingly complex application logics. Not only Web engineers in general but also specifically Web frameworks engineers bias client-side processing of application functionality to server-side processing. This seems the reason for many of them not to regard the database technology as a wide-ranging consistency handler, other than its storage role. For the sake of performance, one might prefer and come with an alternative solution to the problem, client-side dispensation of spatial constraints by engaging open source libraries into the game. Some of these libraries are also serving the same purpose by being included inside spatial databases, such as the GEOS module. Therefore, the trade-off is between choosing a self-contained database or better performance. It is also important not to forget that both sides of processing have their own respective benefits.

8.3.5 Realize Spatial enrichments to OCL

Previously, we claimed OCL is model-level standard syntax for specification and implementation of constraints. Afterwards, the question that seeks a special treatment is whether it supports spatial constraints or not. Section 3.6 cataloged some recent studies conducted under such theme. The main question remained, “how consistent the database at the back of the Web application could be?”, given the short time
allotted for this MSc thesis. Therefore, manual specification of the constraints was believed to dish up our intention, and has been embedded onto our architectural re-engineering. If it was not about time, spatialized-OCL would have automated the entire UWE-led design process (of Chapter 5), by assimilating constrains as main part of the Content model and its transformations.

8.3.6 Enhance the current functionality of our WFS-T prototype

There is still room to upgrade our consistent WFS-T prototype. We prioritized the examination of constraints at different level of abstraction and their enforcement at different level of complexity. The intention was, therefore, to put at least one case that demonstrates the situation at each segregated level. This takes us to the conclusion that our list of constraints and functions was not full-blown enough to model all constraints and functions in our use-case. There are some details left out during the implementation of the prototype. Like, we may add a function that fills existing boundary slivers of the distribution geometry or slivers that emerge as a result of merging interfluvial boundary while we restrict them into species distribution geometry (\textit{restrict\_distrib} function, Section 5.2). Again, we may also constrain the maximum size extent of the slivers for removal (they will be left intact otherwise).

8.3.7 Utilize extended version of UML for MVC modelling

One of the most distinguishing features of UML as a modeling tool is its wider applicability owing to its built-in extensibility mechanism to satisfy modeler’s flexible needs to define and use their own domain-specific elements, called Domain Specific Language (DSL), beyond the already defined basics. For instance, UML does not have a built-in construct to model the Web-specific systems or MVC triad; therefore, one needs to extend the UML notation language to incorporate the specifics. The extensions of UML meta-model happens by defining a set of stereotypes, tagged values and constraints, and compile it in a so called UML profile (which is in a more technical terminology called UML “lightweight” or “non-intrusive” extension)[66]. This way, modeling of special levels of Web applications defined above (like, for instance, the navigation and presentation pieces) are possible by graphically representing them using UML’s extension mechanisms[15].

We are pointing our fingers to two prototypes that have been developed by[57] and[19] on how to extend the UML to adapt the WE modeling approaches, integrate UML with the MVC triad. They are just propositions not yet implemented by any UML tool. Here, we are specifying a recommendation that we believe might aggravate the efficacy of application modeling around MVC-led Web frameworks.

8.3.8 Put a powerful wrapper for the creation of useful database functions, callable from the Web later

During the prototype development, the new Web frameworks architecture called for the creation of both simple and complex database functions which are callable right from the controller. For some of the functions that dictate the application logic of our WFS-T prototype, we provided not only a ‘function-caller wrapper’ that calls them from the application side, but also a ‘function-creator wrapper’ that creates them as part of the data modeling inside the ‘Model’ (the M). For those function which require a bit complicated syntax, it was intricate, if not impossible, to make a timely enrichment of the ‘function-creator wrapper’ during the MSc period. They were defined on the server-side, using a database client of our preferred platform (the pgAdmin III of PostgreSQL). Therefore, we recommend a further look at the completion of the initiative to provide a ‘function-creator wrapper’ inside the framework itself. The good reason for this could be that it exactly matches the common practices around Web frameworks engineering and it helps developers concentrate on other important matters than unnecessarily struggling to know SQL. However, for database administrators that seek to make advantage of the Web in some ways, this problem may not be counted.

8.3.9 Comprise exception handlers inside Web frameworks

In our web2py application logics coded using python for our prototype, the approach to catch exception upon failure of execute commands sent to the database was by making use of the pythonic ‘\textit{try\_except}’ statement. If the operation under the \textit{try} block could not be successfully executed, the action indicated
under the `except` block gets the turn, usually in the form of error report. In this routine, it is found trouble-free to grab hold of the values that the user-defined database functions return. And, it was possible to use them to inform the Web user about what is happening under-the-hood. But, in cases when Check constraints or Trigger functions were employed, it was found unsupported by the connectivity of web2py to report the kind of intermediate error that it stumbled upon in the middle of the execution. We are specifying a recommendation here about the inclusion of a striking error handler that unerringly returns the kind of error encountered, in stead of merely notifying the occurrence of unidentified error.

### 8.3.10 Make a full-fledged WDA design

Our design process in Chapter 5 used the EA modeling tool. We applied use-case and activity UML diagrams to model the Requirements and UML Class diagram to model the Content level. The design issue of the Navigation and Presentation levels was intentionally excluded. This is not because we think it is trivial but because we did not want to contemplate on a design issue that has less to do with the consistency dealing of our project. For that, EA was found sufficiently equipped UML tool. However, we are aware of the reality that there are many open source and commercial tool out in the market essentially designed for a full-fledged UWE-led Web application modeling, such as the ArgoUML (http://argouml.tigris.org/) and the MagicDraw (http://www.magicdraw.com/). The use of such tools would have helped to fully materialize the principles of UWE and MDA.

### 8.3.11 Intensify the current guideline for spatialization of Web frameworks

The set of instructions we developed in Chapter 4 to make a spatial enhancement of Web frameworks has been examined to successfully build a standard spatial Web application powered by web2py. This Web service is the first of its kind not only because it is designed for a consistent and self-sufficient database but also because it is the first spatial application in web2py’s history. That being accredited, it is completely out of truth if we say the guideline of spatialization has been superlatively designed and is conclusive. It leaves a lot of significant details. The main target has been, however, met, for it has been proved to be a valuable and helpful director for a preliminary spatialization of Web frameworks.

### 8.3.12 Create a ORMapper from SQL into web2py M

This is an extra task that might automate and enhance the effectiveness of the design process. This is a simple example of what design practitioners call ‘Reverse Engineering’. This is a broad concept and we don’t want to import it here. But, the idea is, ORM in Web frameworks engineering were seen from as uni-directional only, generation of SQL codes from objects encapsulated inside the framework. Inversion of this practice is also possible and helpful. This is believed to make developers worry less about unnecessary details and concentrate on more important topics, such as constraints handling.

### 8.4 The Impact on other Web frameworks

We don’t want to put up the shutters without addressing the implication of the study on frameworks other than web2py. Web2py has been made to represent MVC-founded Web frameworks. Did it really do that? Do all our dealings about Web frameworks by particularly targeting web2py really influence the rest of them? The impacts can be discussed in terms of three views.

#### 8.4.1 In terms of Modeling

Modeling being the lately introduced approach by this MSc project to Web frameworks world, its deduction is not only to ease spatial database constraints management but also to bring the culture of WE into them. This enables the design and development of model-led across-the-board Web applications with the help of Web frameworks. Other Web frameworks can still utilize the same modeling procedure. Our Chapter 2 is the root resource here; it at a glimpse reviews the literature on design-driven modeling via standard notational and visual languages for robust, data-intensive and maintainable Web applications. Plus, they can also refer Chapter 5 for practical illustration; it partially applies the method under the design process of a prearranged use-case.
Tools, approaches and platforms picked in our case are matters of choice. This refers to subjective preferences of spatial database, Web framework, UML tool and WE method. Apart from that, the underlying fact remains alike and could be inherited intact by other Web frameworks. There are still enough rooms for evolvement of the procedure by incorporating own philosophies into it. Hence, we believe this study is outstanding and influential not only specifically to web2py but also to all other MVC-led frameworks.

8.4.2 In terms of Consistency

Spatial database consistency is not a topic for many applications implemented on a Web platform. We also know that web frameworks are not exception to this. But lately, the consistency frame we designed in this project is believed to affect not only Web2py and the rest of Web frameworks but also other spatial Web development tools that rely on shallow client-side control of few inconsistency-resulting operations. Nevertheless, we already admitted that there could also exist some other alternatives for an even advance and robust control of database consistency, in Section 8.3.1.

Few minutes of googling proves the value we added into the spatial Web culture, more than ever with regard to consistency control. Sometimes, there is nothing that controls you as an anonymous and unauthorized visitor of the Web application page from making invalid insertions or modification of valuable data. If needed, let us base our claim on a practical example by making a closer look at this page (http://dev4.mapgears.com/bdga-mapfish/bdgaWFS-T.html).

8.4.3 In terms of Spatialization

Web frameworks, welcome to the spatial world!!! If there is a good news this thesis work brought into the mainstream of the non-spatial Web frameworks, it is this preliminary guideline we devised and experimented on web2py in Chapter 4, to assist us deal with the remaining subjects of the project that follow it. The guideline has been tested to be as much as necessary for a foundation and starting point. However, a call for its further intensification were forwarded in the recommendation section of the current chapter (see 8.3.11).
8.4. The Impact on other Web frameworks
Bibliography


[21] Panagiotis (Peter) Vretanos (Editor) Douglas Nebert (Editor), Arliss Whiteside (Editor). OpenGIS® catalogue services specification. 2007.


Bibliography


Appendix A

Extending the M of web2py

A.1 Target geometric fields

We said, the first step in spatializing the M of Web frameworks is to identify the geometry structure we want them to support. For the moment, we aspire to launch a spatialized version of web2py by making it support six simple but standard geometry types as recognized by OGC®, shown in Figure 3.2 - POINT, LINESTRING, POLYGON and their MULTI batches. It is not impossible to add more complicated types later.

To make these basic geometric data types recognized by web2py, we appended a hash of strings into the ‘translator’ module inside the DAL that maps web2py objects into constructs and SQL dialects native to each database drivers (PostgreSQL in our case). To further grow the existing set of supported data types, the first pace we made was including new mapper into the translator layer, the same way it is roughly shown here.

```python
SQL_DIALECTS = {
    ...
    'postgres': {
        ...
        'point': 'POINT',
        'POINT': 'POINT',
        'multipoint': 'MULTIPOINT',
        'MULTIPOINT': 'MULTIPOINT',
        'line': 'LINESTRING',
        'LINE': 'LINESTRING',
        'linestring': 'LINESTRING',
        'LINESTRING': 'LINESTRING',
        'multiline': 'MULTILINESTRING',
        'MULTILINESTRING': 'MULTILINESTRING',
        'multilinestring': 'MULTILINESTRING',
        'MULTILINESTRING': 'MULTILINESTRING',
        'polygon': 'POLYGON',
        'POLYGON': 'POLYGON',
        'multipolygon': 'MULTIPOLYGON',
        'MULTIPOLYGON': 'MULTIPOLYGON',
    },
    ...
}
```
A.2 The $gField$ initializer

As we have identified earlier, the second step in our guideline to spatialize Web frameworks is to create a wrapper that initiates the geometry column object. This is the wrapper which a user or a developer calls (see how it is used in Section 4.3.2). At the next step, another wrapper as shown in Section A.3 adds it into its parent table. However, the action performed by the second wrapper is hidden from the user or developer. We called the wrapper that users could apply to define their target geometry column as the $gField$. This newly introduced class tags the grammar of supported geometric fields, listed in Section A.1. It should be initialized with a name, type, SRID and dimensions as follows. The other two initialized values ($index$ (boolean) and ‘within’) are unimportant at this level and are explained later.

```
class gField(object):
    def __init__(self,
        gfname,
        gType,
        SRID=4326,
        dim=2,
        index=True,
        within=()):
        (self.name,
         self.type,
         self.SRID,
         self.dim,
         self.index,
         self.within)=(gfname, gType, SRID, dim, index, within)
```

A.3 The $gField$ creator

The block of code shown below, with the name `spatializer`, is our internal wrapper that acts on already created parent table and adds a geometry column into it. This is the third step in the process of spatializing the M. It is utterly not self-sufficient in the logic that there are some more lines of codes scattered within the DAL file to make it fully functional function. These strewn lines will not be explicated here because they are believed to unnecessarily confuse the reader.

What is it doing? In simplistic manner, what it is performing is, it receives objects which are recognized as instances of $gField$, translates the geometry datatype into a native dialect of the database server (PostgreSQL in this case), add the geometry column into its parent table if there is no geometry column that assumes identical name and keep a record about it for a later use.

```
def spatializer(self):
    for qq in self.gFields:
        gF = self[qq]

        if not gF.type in self._db._translator:
            # //non-gField type\\
            raise SyntaxError, 'Unknown GEOMETRY field type: %s for %s' % 
                (gF.type, gF.name)
        else:
            gType = self._db._translator[gF.type]

        tfile = open(self._dbt, 'r')
        portalocker.lock(tfile, portalocker.LOCK_SH)
```
Appendix A. Extending the M of web2py

A.4 Wrapper for invoking server-side functions in web2py

The benefit of calling stored or defined server-side functions is two fold. First, it is forwarded as an option in the enhancement of the C of MVC in Section 4.3.3. Secondly, it is recommended in Section 3.6.1 for an advanced spatial database consistency. Hence, its access and usage from the Web application layer is mandatory. The wrapper we created inside web2py looks like the following. Its usage is illustrated in Section 4.3.3.

```python
def fire(self, finc):
    self.func = finc
```
fiQuery = 'SELECT %s' % finc

resTuple = self.executesql(fiQuery)
self.commit()
for j in resTuple:
    return j[0]
Appendix B

Constraints handlers

B.1 Constructor of Checks in web2py

This is a Check constraint handler we introduced into web2py, written in python. It gives the user a chance to define own check constraints by specifying the ‘check expressions’ and assigning it to the check attribute of the attribute or table for which the check holds. See Section 6.2.1 for more discussion and example on this topic.

def checkerizer(self, checker=[]):
    for check in checker:
        if self._dbt:
            tfile = open(self._dbt, 'r')
            portalocker.lock(tfile, portalocker.LOCK_SH)
            gChe = cPickle.load(tfile)
            portalocker.unlock(tfile)
            tfile.close()
            if not check in gChe['gChecks'].values():
                self.gCheckers.append(check)

            iii = len(gChe['gChecks']) + 1
            che = 'web2py_checker%s' % (iii)

            cheuery = "ALTER TABLE %s ADD CONSTRAINT %s CHECK (%s);" %
                        (self._tablename, che, check)

            self._db['_lastsql'] = cheuery
            self._db._execute(cheuery)
            self._db.commit()

            logfile = open(self._logfile, 'a')
            logfile.write(\"\n' + cheuery + "\n" CHECK CONSTRAINT ('%s') added to TABLE: %s'\n" %
                        (che, self._tablename))
            logfile.close()

            gChe['gChecks'][che] = check

        tfile = open(self._dbt, 'w')
        portalocker.lock(tfile, portalocker.LOCK_EX)
Appendix B. Constraints handlers

B.2 Constructor of within trigger in web2py

def triggerizer(self):
    for qF in self.gFields:
        gF = self[qF]

        if self._dbt and gF.within:
            wG = gF.within[0]
            wT = gF.within[1]
            aSelf = gF.name

            tfile = open(self._dbt, 'r')
            portalocker.lock(tfile, portalocker.LOCK_SH)
            gTri = cPickle.load(tfile)
            portalocker.unlock(tfile)
            tfile.close()

            tr = 'web2py_trigger_%swithin%s()' % (gF.name, wG)
            tt = 'web2py_trigger_%swithin%s' % (gF.name, wG)

            if not tt in gTri['gTriggers'].values():
                self.gTriggers.append(tt)

                withinQ = '''SELECT ST_WITHIN(NEW.%s, %s.%s) FROM %s AS %s'''
                wErr = '%s should be Within %s Geometry of %s' % (gF.name, wG)

                triQuery = "CREATE OR REPLACE FUNCTION %s
                AS
                $BODY$
                BEGIN
                    IF (%s) = 'f' THEN
                        RAISE EXCEPTION
                    ELSE IF (%s) = 'f' THEN
                        RAISE EXCEPTION
                    END IF;
                RETURN NEW;
                END;
                $BODY$
                LANGUAGE
                'plpgsql';
                CREATE TRIGGER %s BEFORE INSERT OR UPDATE ON
                %s
                FOR EACH ROW EXECUTE PROCEDURE %s;
                %
                (tr, withinQ, wErr, tt, self._tablename, tr)

            self._db[]._lastsql = triQuery
            self._db._execute(triQuery)
            self._db.commit()

            logfile = open(self._logfile, 'a')
            logfile.write('"%s" added to TABLE: %s
            (tt, self._tablename))
            logfile.close()

            gTri['gTriggers'][tt] = tt

            tfile = open(self._dbt, 'w')
portalocker.lock(tfile, portalocker.LOCK_EX)
cPickle.dump(gTri, tfile)
portalocker.unlock(tfile)
tfile.close()

B.3 Server-side functions

These following database functions are some of the callables triggered by the Web application for two purposes. One, to accommodate as much constraints as possible in terms of the notion introduced in Section 3.6.1. Two, to enhance the controller for user-dependent application logic. The inclusion of constraints is eXtensible, the is still room a later addition of nested blocks of code into these functions. Thus, what we have are showing here are only at an embryo stage. The geometry inputs readable by these functions are columns are of WKT in format.

--to ONE GEOMETRY
DROP FUNCTION restrict_distrib (idfier integer, geom1 TEXT);

CREATE OR REPLACE FUNCTION restrict_distrib (idfier integer, geom1 TEXT)  
    RETURNS BOOLEAN AS
$BODY$
BEGIN
    UPDATE distribution
    SET distribgeom=ST_MULTI(
        ST_GeomFromText($2, 4326))
    WHERE cidx=$1;
    IF FOUND THEN
        RETURN TRUE;
    ELSE
        RETURN FALSE;
    END IF;
END;
$BODY$
    LANGUAGE 'plpgsql' VOLATILE

--to TWO GEOMETRIES
DROP FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT);

CREATE OR REPLACE FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT)  
    RETURNS BOOLEAN AS
$BODY$
BEGIN
    UPDATE distribution
    SET distrib=ST_MULTI(ST_UNION(ARRAY[
        ST_GeomFromText($2, 4326),
        ST_GeomFromText($3, 4326)]))
    WHERE cidx=$1;
    IF FOUND THEN
        RETURN TRUE;
    ELSE

115
RETURN FALSE;
END IF;

END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE

--to THREE GEOMETRIES
DROP FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT);
CREATE OR REPLACE FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT)
  RETURNS BOOLEAN AS
$BODY$
BEGIN
  UPDATE distribution
  SET distrib=ST_MULTI(ST_UNION(ARRAY[
    ST_GeomFromText($2, 4326),
    ST_GeomFromText($3, 4326),
    ST_GeomFromText($4, 4326)]))
  WHERE cidx=$1;
  IF FOUND THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE

--to FOUR GEOMETRIES
DROP FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT);
CREATE OR REPLACE FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT)
  RETURNS BOOLEAN AS
$BODY$
BEGIN
  UPDATE distribution
  SET distrib=ST_MULTI(ST_UNION(ARRAY[
    ST_GeomFromText($2, 4326),
    ST_GeomFromText($3, 4326),
    ST_GeomFromText($4, 4326),
    ST_GeomFromText($5, 4326)]))
  WHERE cidx=$1;
  IF FOUND THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE

--to FIVE GEOMETRIES
DROP FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT, geom5 TEXT);
CREATE OR REPLACE FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT, geom5 TEXT)
RETURNS BOOLEAN AS
$BODY$
BEGIN
  UPDATE distribution
  SET distrib=ST_MULTI(ST_UNION(ARRAY[
    ST_GeomFromText($2, 4326),
    ST_GeomFromText($3, 4326),
    ST_GeomFromText($4, 4326),
    ST_GeomFromText($5, 4326),
    ST_GeomFromText($6, 4326)]))
  WHERE cidx=$1;
  IF FOUND THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE

--to SIX GEOMETRIES
DROP FUNCTION restrictTo(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT, geom5 TEXT, geom6 TEXT);
CREATE OR REPLACE FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT, geom5 TEXT, geom6 TEXT)
RETURNS BOOLEAN AS
$BODY$
BEGIN
  UPDATE distribution
  SET distrib=ST_MULTI(ST_UNION(ARRAY[
    ST_GeomFromText($2, 4326),
    ST_GeomFromText($3, 4326),
    ST_GeomFromText($4, 4326),
    ST_GeomFromText($5, 4326),
    ST_GeomFromText($6, 4326),
    ST_GeomFromText($7, 4326)]))
  WHERE cidx=$1;
  IF FOUND THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
END;
$BODY$
--to SEVEN GEOMETRIES
DROP FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT, geom5 TEXT, geom6 TEXT, geom7 TEXT);

CREATE OR REPLACE FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT, geom5 TEXT, geom6 TEXT, geom7 TEXT)
RETURNS BOOLEAN AS
$BODY$
BEGIN
UPDATE distribution
SET distrib=ST_MULTI(ST_UNION(ARRAY[
ST_GeomFromText($2, 4326),
ST_GeomFromText($3, 4326),
ST_GeomFromText($4, 4326),
ST_GeomFromText($5, 4326),
ST_GeomFromText($6, 4326),
ST_GeomFromText($7, 4326)]))
WHERE cidx=$1;
IF FOUND THEN
RETURN TRUE;
ELSE
RETURN FALSE;
END IF;

END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE

--to EIGHT GEOMETRIES
DROP FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT, geom5 TEXT, geom6 TEXT, geom7 TEXT, geom8 TEXT);

CREATE OR REPLACE FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT, geom5 TEXT, geom6 TEXT, geom7 TEXT, geom8 TEXT)
RETURNS BOOLEAN AS
$BODY$
BEGIN
UPDATE distribution
SET distrib=ST_MULTI(ST_UNION(ARRAY[
ST_GeomFromText($2, 4326),
ST_GeomFromText($3, 4326),
ST_GeomFromText($4, 4326),
ST_GeomFromText($5, 4326),
ST_GeomFromText($6, 4326),
ST_GeomFromText($7, 4326),
ST_GeomFromText($8, 4326)]))
WHERE cidx=$1;
IF FOUND THEN
RETURN TRUE;
RETURN FALSE;
END IF;

END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE
ELSE
  RETURN FALSE;
END IF;

END;

$BODY$
  LANGUAGE 'plpgsql' VOLATILE

--to EIGHT GEOMETRIES
DROP FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT, geom5 TEXT, geom6 TEXT, geom7 TEXT, geom8 TEXT);

CREATE OR REPLACE FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT, geom5 TEXT, geom6 TEXT, geom7 TEXT, geom8 TEXT)
  RETURNS BOOLEAN AS
$BODY$
BEGIN
  UPDATE distribution
  SET distrib=ST_MULTI(ST_UNION(ARRAY[
    ST_GeomFromText($2, 4326),
    ST_GeomFromText($3, 4326),
    ST_GeomFromText($4, 4326),
    ST_GeomFromText($5, 4326),
    ST_GeomFromText($6, 4326),
    ST_GeomFromText($7, 4326),
    ST_GeomFromText($8, 4326),
    ST_GeomFromText($9, 4326)]))
  WHERE cidx=$1;
  IF FOUND THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
END;

$BODY$
  LANGUAGE 'plpgsql' VOLATILE

--to NINE GEOMETRIES
DROP FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT, geom5 TEXT, geom6 TEXT, geom7 TEXT, geom8 TEXT, geom9 TEXT);

CREATE OR REPLACE FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT, geom5 TEXT, geom6 TEXT, geom7 TEXT, geom8 TEXT, geom9 TEXT)
  RETURNS BOOLEAN AS
$BODY$
BEGIN
  UPDATE distribution
  SET distrib=ST_MULTI(ST_UNION(ARRAY[
    ST_GeomFromText($2, 4326),
    ST_GeomFromText($3, 4326),
    ST_GeomFromText($4, 4326),
    ST_GeomFromText($5, 4326),
    ST_GeomFromText($6, 4326),
    ST_GeomFromText($7, 4326),
    ST_GeomFromText($8, 4326),
    ST_GeomFromText($9, 4326)]))
  WHERE cidx=$1;
  IF FOUND THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
END;

$BODY$
  LANGUAGE 'plpgsql' VOLATILE
Appendix B. Constraints handlers

```
--to TEN GEOMETRIES
DROP FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT, geom5 TEXT, geom6 TEXT, geom7 TEXT, geom8 TEXT, geom9 TEXT, geom10 TEXT);

CREATE OR REPLACE FUNCTION restrict_distrib(idfier integer, geom1 TEXT, geom2 TEXT, geom3 TEXT, geom4 TEXT, geom5 TEXT, geom6 TEXT, geom7 TEXT, geom8 TEXT, geom9 TEXT, geom10 TEXT) RETURNS BOOLEAN AS
$BODY$
BEGIN
  UPDATE distribution
  SET distrib=ST_MULTI(ST_UNION(ARRAY[
    ST_GeomFromText($2, 4326),
    ST_GeomFromText($3, 4326),
    ST_GeomFromText($4, 4326),
    ST_GeomFromText($5, 4326),
    ST_GeomFromText($6, 4326),
    ST_GeomFromText($7, 4326),
    ST_GeomFromText($8, 4326),
    ST_GeomFromText($9, 4326),
    ST_GeomFromText($10, 4326)]))
  WHERE cidx=$1;
  IF FOUND THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE

CREATE OR REPLACE FUNCTION add_distrib (cidxx numeric, geom text) RETURNS text AS
$BODY$
BEGIN
  UPDATE distribution
  SET distrib=ST_MULTI(ST_UNION(ARRAY[
    ST_GeomFromText($2, 4326),
    ST_GeomFromText($3, 4326),
    ST_GeomFromText($4, 4326),
    ST_GeomFromText($5, 4326),
    ST_GeomFromText($6, 4326),
    ST_GeomFromText($7, 4326),
    ST_GeomFromText($8, 4326),
    ST_GeomFromText($9, 4326),
    ST_GeomFromText($10, 4326),
    ST_GeomFromText($11, 4326)]))
  WHERE cidx=$1;
  IF FOUND THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE
```

CREATE OR REPLACE FUNCTION add_distrib (cidxx numeric, geom text) RETURNS text AS
$BODY$
BEGIN
```
INSERT INTO distribution (cidx, distrib) VALUES ($1, ST_GeomFromText($2, 4326));
IF FOUND THEN
RETURN TRUE;
ELSE
RETURN FALSE;
END IF;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE

CREATE OR REPLACE FUNCTION update_distrib (idfier integer, idx integer, geom text)
RETURNS text AS
$BODY$
BEGIN
UPDATE distribution
SET cidx=$2, distrib=ST_GeomFromText($3, 4326)
WHERE gid=$1;
IF FOUND THEN
RETURN 'success!!!';
ELSE
RETURN 'failure!!!';
END IF;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE
Appendix B. Constraints handlers
Appendix C

Phases of content model

C.1 Logical database model

Figure C.1 depicts the one-step transformation of the content model shown in Figure 5.8 from the conceptual to the still PIM logical model. This model will be later mapped to a PSM representation of the spatial database.

![Logical database model](image)
Appendix C. *Phases of content model*

C.2 Physical database model

Here follows the PSM as generated directly from the above logical database model. PRIMARY KEY, FOREIGN KEY, DATA TYPE, UNIQUE (and NOT NULL), GEOMETRY columns creation are some of the primitives which need further modification, to fit the purpose and platform we have in mind.

```sql
CREATE TABLE Family (
    famname varchar(50),
    fidx int,
    taxorder short,
    familyID integer NOT NULL);

CREATE TABLE Genus (
    genusname varchar(50),
    gidx int,
    taxorder short,
    genusID integer NOT NULL,
    some_genus integer NOT NULL);

CREATE TABLE Species (
    authority varchar(50),
    cidx int,
    eng_name varchar(50),
    listname varchar(50),
    port_name varchar(50),
    taxorder short,
    speciesID integer NOT NULL,
    some_species integer);

CREATE TABLE Distribution (
    distrib GM_MultiSurface,
    distributionID integer NOT NULL,
    my_distrib integer);

CREATE TABLE AmazoniaLegal (
    geom GM_Polygon,
    id int,
    buffer GM_Polygon,
    amazoniaLegalID integer NOT NULL);

CREATE TABLE Interfluve (
    fid int,
    fluvename varchar(50),
    geom GM_Polygon,
    inamazonia boolean,
    interfluveID integer NOT NULL);

ALTER TABLE Family ADD CONSTRAINT PK_Family
    PRIMARY KEY (familyID);

ALTER TABLE Genus ADD CONSTRAINT PK_Genus
    PRIMARY KEY (genusID);

ALTER TABLE Distribution ADD CONSTRAINT PK_Distribution
    PRIMARY KEY (distributionID);
```
Appendix C. Phases of content model

ALTER TABLE Species ADD CONSTRAINT PK_Species
    PRIMARY KEY (speciesID);

ALTER TABLE AmazoniaLegal ADD CONSTRAINT PK_AmazoniaLegal
    PRIMARY KEY (amazoniaLegalID);

ALTER TABLE Interfluve ADD CONSTRAINT PK_Interfluve
    PRIMARY KEY (interfluveID);

ALTER TABLE Distribution ADD CONSTRAINT specDistrib
    FOREIGN KEY (my_distrib) REFERENCES Species (speciesID);

ALTER TABLE Genus ADD CONSTRAINT hasFamily
    FOREIGN KEY (some_genus) REFERENCES Family (familyID);

ALTER TABLE Species ADD CONSTRAINT hasGenus
    FOREIGN KEY (some_species) REFERENCES Genus (genusID);

C.3 Physical database model: edited version

Few manual editions are made to the PSM physical model shown above. This was required in order to come with directly deployable code on a PostgreSQL, our chosen database platform. It is this final model that gets mapped into web2py M for database construction that represents the content model. There is no ready to use mapper (see Section 8.3.12). Hence, we made the transformation manually. The rationale for this edition are listed in Section 6.2. See the comment line at the end of each SQL statements (marked with double ‘-’ character) for the added constraints.

CREATE TABLE Family (
    famname varchar(50) UNIQUE,
    fidx int,
    taxorder short,
    familyID integer NOT NULL);
--added UNIQUE constraint to famname >> Constraint1

CREATE TABLE Genus (
    genusname varchar(50) UNIQUE,
    gidx int,
    taxorder short,
    genusID integer NOT NULL,
    some_genus integer NOT NULL);
--added UNIQUE constraint to genusname >> Constraint1

CREATE TABLE Species (
    authority varchar(50),
    cidx int,
    eng_name varchar(50) PRIMARY UNIQUE NOT NULL,
    listname varchar(50) PRIMARY UNIQUE NOT NULL,
    port_name varchar(50) PRIMARY UNIQUE NOT NULL,
    taxorder short,
    speciesID integer NOT NULL,
    some_species integer);
--added PRIMARY KEY, UNIQUE and NOT NULL constraints to eng_name, listname and
Appendix C. Phases of content model

-- port_name >> Constraint2 + Constraint7

CREATE TABLE Distribution (  
distrib GM_MultiSurface,  
distributionID integer NOT NULL,  
my_distrib integer);

ALTER TABLE distribution ADD CONSTRAINT simple_distrib CHECK (ST_issimple(distrib))
-- added ST_issimple, ST_isclosed and ST_isvalid constraints to distrib >> Constraint3
-- Note that, these three check functions are stored GEOS procedures inside PostGIS
-- a trigger for Constraint4 would be added later

CREATE TABLE AmazoniaLegal (  
geom GM_Polygon,  
id int,  
buffer GM_Polygon,  
amazoniaLegalID integer NOT NULL);

CREATE TABLE Interfluve (  
_fid int,  
fluvename varchar(50),  
geom GM_Polygon,  
inamazonia boolean,  
interfluveID integer NOT NULL);

ALTER TABLE Species ADD CONSTRAINT PK_Species  
PRIMARY KEY (speciesID);  
-- These SQLs define Constraint6

ALTER TABLE Genus ADD CONSTRAINT hasFamily  
FOREIGN KEY (some_genus) REFERENCES Family (familyID);  
-- This SQL defines Constraint5

C.4 Web2py model

Here follows the equivalent version of the above edited PSM into web2py M, ‘q’ being the opening of a connection to the database server. The most central part of our discussion in this project, the constraints handling, added few things more, such as the constraints handlers at different levels (marked with the ‘#’ character at the end of each block). The discussion about the implementation of these handlers appears in Section 6.2.1.

q=SQLDB('postgres://postgres:postgres@localhost:5432/habtom')
q.define_table('species',  
SQLField('authority'),  
SQLField('cidx', 'integer'),  
SQLField('eng_name', primary=True, unique=True, notnull=True),  
SQLField('listname', primary=True, unique=True, notnull=True),  
SQLField('port_name', primary=True, unique=True, notnull=True),  
SQLField('taxorder', 'short'))
# added PRIMARY KEY, UNIQUE and NOT NULL constraints to eng_name, listname and
# port_name >> Constraint2 + Constraint7 + Constraint6

q.define_table('family',  
SQLField('famname', unique=True),  
SQLField('authority'),  
SQLField('cidx', 'integer'),  
SQLField('eng_name', primary=True, unique=True, notnull=True),  
SQLField('listname', primary=True, unique=True, notnull=True),  
SQLField('port_name', primary=True, unique=True, notnull=True),  
SQLField('taxorder', 'short'))
# added PRIMARY KEY, UNIQUE and NOT NULL constraints to eng_name, listname and
# port_name >> Constraint2 + Constraint7 + Constraint6
C.5 Intermediary file

Obviously, the standard language to interact with databases is SQL. That is why Web frameworks make a generation of the native dialect of the database beneath. The idea is very simple, the requests they sent would not be listened and responded otherwise. Here comes the need of ORM (see Section 2.5.3), objects at framework layer would be mapped into the equivalent vocabulary of the chosen database. Having this in mind, the following SQL file is the automatically generated version of the above web2py model, including some extra messages. The lines with the ‘>>>’ marks indicate that they are generations of the layers we added into web2py as a result of the spatial enhancement we made in Chapter 4.

```
timestamp: 2010-02-12T19:16:15.578000
CREATE TABLE species(
    port_name VARCHAR(32) PRIMARY KEY NOT NULL NOT NULL UNIQUE,
    taxorder INTEGER
);
success!
```

```
timestamp: 2010-02-12T19:16:15.657000
CREATE TABLE family(
    __table_name__
);
Appendix C. Phases of content model

id SERIAL PRIMARY KEY,
famname VARCHAR(32) UNIQUE,
fidx INTEGER,
taxorder INTEGER
);
success!

timestamp: 2010-02-12T19:16:15.814000
CREATE TABLE genus(
id SERIAL PRIMARY KEY,
some_family INTEGER REFERENCES family(id) ON DELETE CASCADE,
genusname VARCHAR(32) UNIQUE,
gidx INTEGER,
taxorder INTEGER
);
success!

SELECT ADDGEOMETRYCOLUMN('amazonialegal', 'buffer', 4326, 'POLYGON', 2);
>>>GEOMETRY COLUMN ('buffer') added to TABLE ('amazonialegal')

CREATE INDEX web2py_amazonialegal_buffer_gist
ON amazonialegal
USING GIST
(buffer);
>>>GIST-INDEX ('web2py_amazonialegal_buffer_gist') created for GEOMETRY COLUMN: 'buffer', of TABLE: 'amazonialegal'

SELECT ADDGEOMETRYCOLUMN('amazonialegal', 'geom', 4326, 'POLYGON', 2);
>>>GEOMETRY COLUMN ('geom') added to TABLE ('amazonialegal')

CREATE INDEX web2py_amazonialegal_geom_gist
ON amazonialegal
USING GIST
(geom);
>>>GIST-INDEX ('web2py_amazonialegal_geom_gist') created for GEOMETRY COLUMN: 'geom', of TABLE: 'amazonialegal'

timestamp: 2010-02-12T19:16:16.233000
CREATE TABLE interfluve(
id SERIAL PRIMARY KEY,
fid INTEGER,
fluvename VARCHAR(32),
inamazonia CHAR(1)
);
success!

SELECT ADDGEOMETRYCOLUMN('interfluve', 'geom', 4326, 'POLYGON', 2);
>>>GEOMETRY COLUMN ('geom') added to TABLE ('interfluve')
CREATE INDEX web2py_interfluve_geom_gist
ON interfluve
USING GIST
(geom);

>>>GIST-INDEX ('web2py_interfluve_geom_gist') created for GEOMETRY COLUMN:
'geom', of TABLE: 'interfluve'

timestamp: 2010-02-12T19:16:16.467000
CREATE TABLE distribution(
id SERIAL PRIMARY KEY,
cidx INTEGER
);
success!

SELECT ADDGEOMETRYCOLUMN('distribution', 'distrib', 4326, 'MULTIPOLYGON', 2);

>>>GEOMETRY COLUMN ('distrib') added to TABLE ('distribution')

CREATE INDEX web2py_distribution_distrib_gist
ON distribution
USING GIST
(distrib);

>>>GIST-INDEX ('web2py_distribution_distrib_gist') created for GEOMETRY COLUMN:
'distrib', of TABLE: 'distribution'

ALTER TABLE distribution ADD CONSTRAINT web2py_checker1 CHECK (ST_issimple(distrib));

>>>CHECK CONSTRAINT ('web2py_checker1') added to TABLE: 'distribution'

ALTER TABLE distribution ADD CONSTRAINT web2py_checker2 CHECK (ST_isvalid(distrib));

>>>CHECK CONSTRAINT ('web2py_checker2') added to TABLE: 'distribution'

ALTER TABLE distribution ADD CONSTRAINT web2py_checker3 CHECK (ST_isclosed(distrib));

>>>CHECK CONSTRAINT ('web2py_checker3') added to TABLE: 'distribution'

CREATE OR REPLACE FUNCTION web2py_trigger_distribwithinthe_buffer()
RETURNS trigger AS
$BODY$
BEGIN
IF (SELECT ST_WITHIN(NEW.distrib, bu.the_buffer) FROM buffer AS bu) = 'f' THEN
RAISE EXCEPTION 'distrib should be Within the_buffer Geometry of buffer';
END IF;
RETURN NEW;
END;
$BODY$
LANGUAGE 'plpgsql';
CREATE TRIGGER web2py_trigger_distribwithinthe_buffer BEFORE INSERT OR UPDATE
ON distribution
FOR EACH ROW EXECUTE PROCEDURE web2py_trigger_distribwithinthe_buffer();

>>>TRIGGER FUNCTION ('web2py_trigger_distribwithinthe_buffer') added to
TABLE: 'distribution'
Appendix C. Phases of content model
Appendix D

The V as a map interface

The codes that appear in this Section define the interface of our WFS-T prototype. Section D.1 deals with the source of the HTML page. The Section that follow deal with the deployment of OpenLayers for an interactive and service oriented spatial Web application. They are split into section that we believe could let us identify the main segments, other wise they all are part of the same file embedded into <SCRIPT> tag of the HTML page.

D.1 Index file - Basic HTML

What appears below is a mingled file of HTML, JavaScript functionality and web2py codes.

```html
<html>
<head>
<title>{{=response.title or 'response.title'}}</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta name="copyright" content="{{=response.copyright or ''}}" />
<meta name="keywords" content="{{=response.keywords or ''}}" />
<meta name="description" content="{{=response.description or ''}}" />
<meta name="robots" content="index, follow" />
{{include 'web2py_ajax.html'}}
<link rel="stylesheet" media="screen, projection" href="/{{=request.application}}/static/olCSS.css" type="text/css" />
<script src="http://maps.google.com/maps?file=api&v=2.47 & key=ABQIAAAAAnfs7bKE82qgb3Z2Ys-oBT2yXp_ZAY8_ufC3CFXhHE1NvwkxSySz_REEpPq-4WZA270wgbtyR3VcA" type="text/javascript"></script>
<script src="http://dev.openlayers.org/sandbox/adube/openlayers/lib/OpenLayers.js"></script>
<script src="http://openlayers.org/dev/lib/Firebug/firebug.js"></script>
<script type="text/javascript" src="/{{=request.application}}/static/wfst.js"></script>
</head>
<body onload="init()">
<h2>{{=message}}</h2>
<div id="undertabs" style="left: -4px; color: rgb(85, 85, 85);"></div>
```
Appendix D. The V as a map interface

```html
<div class="flash">{{=response.flash or ''}}</div>

<div id="map" class="mapia"></div>

<div id="hiddenFormDiv">
  <div id="formDiv">
    <form id='rForm'>
      {{=p1}} {{=p2}} {{=p3}} {{=p4}} {{=p5}} {{=p6}}
    </form>
    <button id="serializeit" type="button" onclick="plzSerialize()"
title="serialize current form & geometry">SERIALIZE</button>
  </div>
</div>

<div id="info">
  <form id="Qform" method="post" action="/{{=request.application}}/
default/update">
    <input id="uniqueIdentifier" type="hidden" value="None" name="uniqueIdentifier"/>
    <input id="geometria" type="hidden" value="None" name="geometria"/>
    <input id="cidxo" type="hidden" value="None" name="cidxo"/>
    <input id="engo" type="hidden" value="None" name="engo"/>
    <input id="listo" type="hidden" value="None" name="listo"/>
    <input id="porto" type="hidden" value="None" name="porto"/>
    <input id="autho" type="hidden" value="None" name="autho"/>
    <input id="taxo" type="hidden" value="None" name="taxo"/>
    <input id="state" type="hidden" value="None" name="state"/>
    <input id="updater" type="submit" value="UPDATE GEOMETRY"/>
  </form>
</div>

<div id="info2">
  <form id="resform" method="post" action="/{{=request.application}}/
default/restrict">
    <input id="resIdentifier" type="hidden" value="None" name="resIdentifier"/>
    <input id="resGeometria" type="hidden" value="None" name="resGeometria"/>
    <input id="restricter" type="submit" value="RESTRICT DISTRIBUTION"/>
  </form>
</div>

<div id="upload">
  <form id="upform" method="post" action="/{{=request.application}}/
default/shp2pg">
    <h3>Upload a shapefile here:</h3>
    <input id="fiload" type="file" value="Browse" name="fiload"/>
    <input id="browser" type="submit" value="Insert"/>
  </form>
</div>
```

132
Appendix D. The V as a map interface

```html
<!-- HTML code for the genus form -->
</html>

D.2 Map definition

The following OpenLayers lines of code defines the basic map with its initial properties. More properties and controls can be appended later using mapia.addControl(); being the name of the control required. See section 6.3.2 for detail.

```javascript
controls = [new OpenLayers.Control.PanZoomBar(),
            new OpenLayers.Control.Navigation(),
            new OpenLayers.Control.LayerSwitcher(),
            new OpenLayers.Control.OverviewMap(),
            new OpenLayers.Control.MousePosition(),
            new OpenLayers.Control.ScaleLine(),
            new OpenLayers.Control.Attribution()];

mapia = new OpenLayers.Map("map", {
    projection: new OpenLayers.Projection("EPSG:4326"),
    controls: controls
});
```
Appendix D. The V as a map interface

D.3 Base Map

The following code adds the base layer to the map defined above. There are many online sources of base layer maps usable with OpenLayers, among which the following four are picked in our case, one from Metacarta and three from Google.

```javascript
//======================
//BASE LAYER// METACARTA
//======================
basia = new OpenLayers.Layer.WMS("Metacarta base map",
"http://labs.metacarta.com/wms-c/Basic.py?",
{layers: "basic"});

//=====================// BASE LAYER// 3 GOOGLE
//=====================
gooSat = new OpenLayers.Layer.Google( "Google Satellite" ,
{type: G_SATELLITE_MAP,
isBaseLayer: true});
goosMap = new OpenLayers.Layer.Google( "Google Map" ,
{});
goosHybrid = new OpenLayers.Layer.Google( "Google Hybrid" ,
{type: G_HYBRID_MAP} );

D.4 Vector layer

This section of the OpenLayers code enables the Web application load GML-based data on top of the base map. It is vector in format, can be reproduced into other available vector forms. It supports functionalities such as feature edition, styling, filtering and etc.

```javascript
//==============
//VECTOR LAYER//
//==============
//STYLE FIRST//
//--------------
//--------------
var rStyle = new OpenLayers.StyleMap({
strokeColor: "black",
strokeWidth: 1,
strokeOpacity: 0.5,
fillOpacity: 0.5
});

colookup = {
"1": {fillColor:"#0aa989"}
};
rStyle.addUniqueValueRules("default", "gid", colookup);
```
Appendix D. *The V as a map interface*

```javascript
// ---
// ---
// ---
// ---
var rRule = [new OpenLayers.Rule({
  symbolizer: {strokeColor: "red", strokeWidth: 2,
               elseFilter: true
  })];
var selectorRule = [new OpenLayers.Rule({
  symbolizer: {strokeColor: "red",
               strokeWidth: 2,
               strokeOpacity: 0.8,
               fillColor: "red"},
  elseFilter: true
 })];

rStyle.styles["default"].addRules(rRule);
rStyle.styles["select"].addRules(selectorRule);
// ---
// ---

saveStrategy = new OpenLayers.Strategy.Save();
// ---
// ---

// FILTER
var shortScreen = new OpenLayers.Filter.Comparison({
  type: OpenLayers.Filter.Comparison.EQUAL_TO,
  property: "cidx",
  value: document.getElementById("callacies").value,"983" });
// ---
// ---

amaLegal = new OpenLayers.Layer.Vector("Amazonia legal",
{  
  styleMap: rStyle,
  strategies: [new OpenLayers.Strategy.Fixed(), saveStrategy],
  projection: new OpenLayers.Projection("EPSG:4326"),
  protocol: new OpenLayers.Protocol.WFS({
    readFormat: new OpenLayers.Format.GML(),
    outputFormat: "GML2",
    version: "1.0.0",
    url: tinyowsURL,
    featureType: "amazonialegal",
    srsName: "EPSG:4326",
    featurePrefix: "sal",
    featureNS: hosteer,
    geometryName: "legalgeom",
    schema: tinyowsURL + "?service=WFS&version=1.0.0&
              request=DescribeFeatureType&TypeName=amazonialegal"
  })
});
// ---
// ---

interflu = new OpenLayers.Layer.Vector(....
```

---

136
"Interfluves",
{
    styleMap: rStyle,
    strategies: [new OpenLayers.Strategy.BBOX(), saveStrategy],
    projection: new OpenLayers.Projection("EPSG:4326"),
    protocol: new OpenLayers.Protocol.WFS(
        readFormat: new OpenLayers.Format.GML(),
        outputFormat: "GML2",
        version: "1.0.0",
        url: tinyowsURL,
        featureType: "interfluves",
        srsName: "EPSG:4326",
        featurePrefix: "inf",
        featureNS: hosteer,
        geometryName: "intflugeom",
        schema: tinyowsURL + "?service=WFS&version=1.0.0&request=DescribeFeatureType&TypeName=interfluves"
    )
});

// BUFFER
buffer = new OpenLayers.Layer.Vector("5km Buffer",
{
    styleMap: rStyle,
    strategies: [new OpenLayers.Strategy.BBOX(), saveStrategy],
    projection: new OpenLayers.Projection("EPSG:4326"),
    protocol: new OpenLayers.Protocol.WFS(
        readFormat: new OpenLayers.Format.GML(),
        outputFormat: "GML2",
        version: "1.0.0",
        url: tinyowsURL,
        featureType: "buffer",
        srsName: "EPSG:4326",
        featurePrefix: "inf",
        featureNS: hosteer,
        geometryName: "the_buffer",
        schema: tinyowsURL + "?service=WFS&version=1.0.0&request=DescribeFeatureType&TypeName=buffer"
    )
});

// Species distribution
wfsTure = new OpenLayers.Layer.Vector("Species distribution",
{
    styleMap: rStyle,
    strategies: [new OpenLayers.Strategy.BBOX(), saveStrategy],
    projection: new OpenLayers.Projection("EPSG:4326"),
    filter: shortScreen,
    protocol: new OpenLayers.Protocol.WFS(
        readFormat: new OpenLayers.Format.GML(),
        outputFormat: "GML2",
        version: "1.0.0",
        url: tinyowsURL,
        featureType: "species",
        srsName: "EPSG:4326",
        featurePrefix: "inf",
        featureNS: hosteer,
        geometryName: "the_ture",
        schema: tinyowsURL + "?service=WFS&version=1.0.0&request=DescribeFeatureType&TypeName=species"
    )
});
Appendix D. The V as a map interface

url: tinyowsURL,
featureType: "distribution",
srsName: "EPSG:4326",
featurePrefix: "dib",
featureNS: hosteer,
geometryName: "distribgeom",
schema: tinyowsURL + "?service=WFS&version=1.0.0&request=DescribeFeatureType&TypeName=distribution"
});
 });

//mapia.addLayers([gooSat, gooMap, gooHybrid, basia, amaLegal, interflu, buffer, wfsTure]);
//mapia.addLayers([basia, wfsTure1, wfsTure2, wfsTure3]);
mapia.zoomToExtent(bounds);

D.5 Controls and editing tools

This section is also part of the above OpenLayers code. But, this compartment deals more with the editing tools and events associated with them. For full functionality of these tools, some pure JavaScript functions were defined, and are presented in the next Appendix E.

//CONTROLS://
//NAVIGATION
    //isDefault: true,
title: "Pan Map"
});

//INSERT
drawor = new OpenLayers.Control.DrawFeature(
wfsTure, OpenLayers.Handler.Polygon,
{id:"drawor",
    title: "Draw Feature",
displayClass: "olControlDrawFeaturePolygon",
handlerOptions: {multi: true}
}
);
drawor.events.register("featureadded", "", featAdded);

//UPDATE
feaditor = new OpenLayers.Control.ModifyFeature(wfsTure, {
title: "Update Feature",
displayClass: "olControlModifyFeature"
});
wfsTure.events.register("beforefeaturemodified","", onEditorStart);
wfsTure.events.register("featuremodified","", onEdit);
Appendix D. The V as a map interface

//wfsTure.events.register("afterfeaturemodified", ",", onEditorEnd);
feaditor.events.register("activate", ",", onEditorActivated);
feaditor.events.register("deactivate", ",", onEditorDeactivated);
//---------------------------------------------------------
//DELETE
//---------------------------------------------------------
delor = new OpenLayers.Control.DeleteFeature( wfsTure,
{"box":true, title: "Delete Feature"});
delor.events.register("activate", ",", onDelActivated);
delor.events.register("deactivate", ",", onDelDeactivated);
delor.events.register("beforefeaturesdeleted", ",", beforeDel);
wfsTure.events.register("beforefeatureremoved", this,
onBeforeRemoved)
//---------------------------------------------------------
//ZOOMING
//---------------------------------------------------------
zoom2max = new OpenLayers.Control.ZoomToMaxExtent({
map: map, title: "Zoom to maximum extent"
});
var zoomIN = new OpenLayers.Control.ZoomBox({
title: "Zoom in: click/drag"
});
var zoomOUT = new OpenLayers.Control.ZoomBox({
out: true,
displayClass: "olControlZoomBoxOut",
title: "Zoom out: click/drag"
});
//---------------------------------------------------------
//NAVIGATION HISTORY
//---------------------------------------------------------
var navtry = new OpenLayers.Control.NavigationHistory();
mapia.addControl(navtry);
navtry.activate();
navtry.previous.title = "<-- Backward history";
navtry.next.title = "Forward history -->";
//---------------------------------------------------------
//HIGHLIGHT
//---------------------------------------------------------
highlighture = new OpenLayers.Control.HighlightFeature({
layer: wfsTure,
displayPopup: true,
popupOffset: {
  "left": 45,
  "right": 0,
  "top": 35
},
popupTitle: "Species id",
popupSize: new OpenLayers.Size(250,105),
style:{
  strokeColor: "black",
strokeWidth: 2,
Appendix D. The V as a map interface

```javascript
// map interface
{
  strokeOpacity: 0.8,
  fillOpacity: 0.8,
  fillColor: "black", // red
}
mapa.addControl(highlighture);
// SELECTION
selector = new OpenLayers.Control.SelectFeature(wfsTure,
  {
    title: "click to select/ctrl-click to unselect",
    clickout: false, toggle: false,
    multiple: false, hover: false,
    toggleKey: "ctrlKey", // ctrl key removes from selection
    multipleKey: "shiftKey", // shift key adds to selection
    box: true,
    //onSelect: serialize// to serialize
  });
// PANEL
paneel = new OpenLayers.Control.Panel(
  {
    //displayClass: "customEditingToolbar"
  }defaultControl: navigateer);
  paneel.addControls([
    navigateer,
    zoom2max,
    //selector,
    zoomIN,
    zoomOUT,
    drawor,
    feaditor,
    delor,
    navtry.previous,
    navtry.next
  ]);
  mapa.addControl(paneel);
// UPDATE FORMATS
vecFormater();
// REQUESTS
tinyURL = tinyowsURL + "?";
tinyURL += "TypeName=distribution&service=WFS&version=1.0.0&request=DescribeFeatureType";
//alert(tinyURL);
OpenLayers.loadURL(tinyURL, "," this, setAttributes);
```
Appendix E

WFS-T Events (JavaScript)

The OpenLayers functions need several other client-side JS functions to make the process flow complete. Some of these functions are events that trigger as editing tools on the Web application become selected and unselected. Their role is irreplaceable. They are part of the logic. See the following functions that were defined for the effectiveness of our WFS-T prototype developed in Chapter 6.

E.1 JavaScript functions

```javascript
function getSelected(){
    return feaditor.feature; // to-be modified (active) feature
}

function getIntflu(){
    return selector.feature; // to-be modified (active) feature
}

function featAdded(object){
    var QoQ = object.feature;
    QoQ.state = OpenLayers.State.INSERT;
    paneel.activateControl(feaditor);
    feaditor.selectControl.select(QoQ);
}

function onEditorStart(object) {
    var QoQ;
    if (object.geometry){
        QoQ = object;
    } else {
        QoQ = object.feature;
    }
    OpenLayers.Console.log("NOTE: UPDATE started", QoQ.id);
    if(highlighture.feature){
        highlighture.resetFeature();
    }
    addFormPopup(QoQ);
    if(QoQ.state != "Insert"){
```
Appendix E. WFS-T Events

```javascript
parsefeatAttr2Form(QoQ, featAttr);
}

toggleHighlighture('disable');
navigatorer.activate();
);

function onEdit(object) {
    var QoQ;
    if (object.geometry)
        QoQ = object;
    else {
        QoQ = object.feature;
    }
    OpenLayers.Console.log("NOTE: UPDATE took place", QoQ.id);
);

function removeFormPopup(feature) {
    oFormDiv = document.getElementById('formDiv');
    oDiv = document.getElementById('hiddenFormDiv');
    oDiv.appendChild(oFormDiv);
    mapia.removePopup(feature.popup);
    feature.popup.destroy();
    feature.popup = null;
};

function onEditorActivated(){
    toggleHighlighture('enable');
};

function onEditorDeactivated(){
    if (!delor.active){
        toggleHighlighture('disable');
    }
};

function toggleHighlighture(wfsTate) {
    if (wfsTate == "enabled" || wfsTate == "enable" ||
      wfsTate == "activate" || wfsTate == "){
        if (!highlighture.active){
            highlighture.activate();
        }
    }
    else if (wfsTate == "disable" || wfsTate == "disabled" ||
      wfsTate == "deactivate"){
        if (highlighture.active){
            highlighture.deactivate();
        }
    }
    else {
        alert("Error: bad parameter in toggleHighlighture function");
    }
};
```
function addFormPopup(feature) {
  var szHTML = "<div style='font-size:.8em'><h1>Attributes form</h1>
      + "<div id='formInPopup'></div>
      + ">
  var oPopupPos, leftOffset = 45, topOffset = 55, rightOffset=0;

  var mapExtent = mapia.getExtent();
  var nReso = mapia.getResolution();
  var nMapXCenter = mapia.getExtent().getCenterPixel().x;
  var nFeatureXPos = feature.geometry.getBounds().getCenterPixel().x;
  var bLeft = nFeatureXPos >= nMapXCenter;

  if(bLeft){ // popup appears top-left position
    oPopupPos = new OpenLayers.LonLat(mapExtent.left,mapExtent.top);
    oPopupPos.lon += leftOffset * nReso;
  }
  else { // popup appears top-right position
    oPopupPos = new OpenLayers.LonLat(mapExtent.right,mapExtent.top);
    oPopupPos.lon -= rightOffset * nReso;
  }
  oPopupPos.lat -= topOffset * nReso;

  var popup = new OpenLayers.Popup.AnchoredBubble("chicken",
    oPopupPos,
    new OpenLayers.Size(250,300),
    szHTML,
    null, true, onFormPopupClose);
  feature.popup = popup;
  mapia.addPopup(popup);

  oFormDiv = document.getElementById('formDiv');
  oDiv = document.getElementById('formInPopup');
  oDiv.appendChild(oFormDiv);
};

function onFormPopupClose(evt) {
  var QoQ = getSelected();
  if (QoQ){
    removeFormPopup(QoQ);
    document.getElementById('rForm').reset();
    feaditor.selectControl.unselect(QoQ);
    toggleHighlighture("enable");
  }
};

function parsefeatAttr2Form(QoQ, featAttributes) {
  for (var i=0, len=featAttr.length; i<len; i++){
    var Attr = featAttributes[i];
    var szValue = QoQ.attributes[Attr];
    if (szValue){
      document.getElementById(Attr).value = szValue;
    }
  }
}
function setAttributes(response) {
    featType = new OpenLayers.Format.WFSDescribeFeatureType_1_0_0();
    QoQs = featType.read(response.responseText);

    var aItems = new Array();
    aItems = QoQs[0].items;
    var j=0;

    for (i=0; i<aItems.length; i++){
        allAttr[i] = aItems[i].name;
        if (document.getElementById(aItems[i].name)){
            featAttr[j] = aItems[i].name;
            j++;
        }
    }
}
}

function onDelActivated(){
toggleHighlighture('enable');
}

function onDelDeactivated(){
if (!feaditor.active){
toggleHighlighture('disable');
}
}

function onEditorStart(object) {
var QoQ;
if (object.geometry){
    QoQ = object;
}
else {
    QoQ = object.feature;
}

OpenLayers.Console.log("NOTE: UPDATE started", QoQ.id);

if(highlighture.feature){
    highlighture.resetFeature();
}
addFormPopup(QoQ);
//parse before adding popup ke?
if(QoQ.state !="Insert"){
    parsefeatAttr2Form(QoQ, featAttr);
}

toggleHighlighture('disable');
navigateer.activate();
};

function beforeDel(object){
var QoQ;


if (object.geometry){
    QoQ = object;
} else {
    QoQ = object.feature;
}
if(confirm("DELETE? You can't undo this action!!!")){
    document.getElementById('uniqueIdentifier').value = QoQ.fid;
    console.log(document.getElementById('uniqueIdentifier').value);
    QoQ.state = OpenLayers.State.DELETE;
}
function onBeforeRemoved(object){
    if(highlighture.feature == object.feature){
        highlighture.resetFeature();
    }
}
function paintFeat(QoQ, regCla){
    if (regCla == null){
        regCla = document.getElementById("id").value;//value of the selected
    }
    wfsTure.drawFeature(QoQ, coLookup[regCla]);
}
//SERIALIZER FUNCTIONS//
function vecFormater(){
    var gOptions = {
        'internalProjection': mapia.baseLayer.projection,
        'externalProjection': mapia.baseLayer.projection
    };
    vecFormats = {
        wkt: new OpenLayers.Format.WKT(gOptions),
        geojson: new OpenLayers.Format.GeoJSON(gOptions)
    };
    function plzSerialize() {
        var QoQ = getSelected();
        if(QoQ.state != OpenLayers.State INSERT) {
            QoQ.state = OpenLayers.State.UPDATE;
        }
        if (QoQ.layer.protocol.CLASS_NAME == "OpenLayers.Protocol.HTTP" ||
            QoQ.layer.protocol.CLASS_NAME == "OpenLayers.Protocol.WFS.v1_0_0"){
            var callableGeom = vecFormats['wkt'].write(QoQ);
            document.getElementById('uniqueIdentifier').value = QoQ.fid;
        }
Appendix E. WFS-T Events

```
//console.log(QoQ.fid);
document.getElementById('geometria').value = callableGeom;
//console.log(document.getElementById('geometria').value);

//console.log(document.getElementById('cidxo').value);
document.getElementById('cidxo').value = document.getElementById('cidx').value;

//console.log(document.getElementById('engo').value);
document.getElementById('engo').value = document.getElementById('eng_name').value;

//console.log(document.getElementById('listo').value);
document.getElementById('listo').value = document.getElementById('listname').value;

//console.log(document.getElementById('porto').value);
document.getElementById('porto').value = document.getElementById('port_name').value;

//console.log(document.getElementById('autho').value);
document.getElementById('autho').value = document.getElementById('authority').value;

//console.log(document.getElementById('taxo').value);
document.getElementById('taxo').value = document.getElementById('taxorder').value;

if (QoQ.geometry){
    QoQ = QoQ;
} else {
    QoQ = QoQ.feature;
}

removeFormPopup(QoQ);
document.getElementById('rForm').reset();
feaditor.selectControl.unselect(QoQ);
toggleHighlighture('enable');
OpenLayers.Console.log("u end modifying", QoQ.id);
}

else {
    alert("unsuported protocol!!!");
}

//FILTER
function plzFilter(){
callacies = document.getElementById('callacies').value
//var conKey = ["eng_name", "listname", "port_name", "authority", "taxorder"];
document.getElementById('filterID').value = callacies
idfier = document.getElementById('filterID').value
var shortScreen = new OpenLayers.Filter.Comparison({
type: OpenLayers.Filter.Comparison.EQUAL_TO,
property: 'cidx',
value: idfier//'983'
});
wfsTure.filter = shortScreen;

wfsTure.refresh({force: true});
}

var resGeometria = "";
function plzRestrict(ture){
var QoQ = ture;
//OpenLayers.Console.log("selected", ture.id);
//console.log(QoQ);
var callableGeom = vecFormats['wkt'].write(QoQ);
//console.log(callableGeom);
//resGeometria.push(callableGeom);
resGeometria+=callableGeom + "/";
document.getElementById('resGeometria').value = resGeometria;
```
console.log(resGeometria);
document.getElementById('resIdentifier').value = document.getElementById('callacies').value;
console.log(document.getElementById('resIdentifier').value);
};

function showFForm() {
document.getElementById("hidFam").style.display = "block";
document.getElementById("hidFam").style.left = "5px";
document.getElementById("hidFam").style.top = "90px";
document.getElementById("hidFam").style.width = "320px";
}

function closeFForm() {
document.getElementById("hidFam").style.display = "none";
}

function showGForm() {
document.getElementById("hidGen").style.display = "block";
document.getElementById("hidGen").style.left = "5px";
document.getElementById("hidGen").style.top = "90px";
document.getElementById("hidGen").style.width = "320px";
}

function closeGForm() {
document.getElementById("hidGen").style.display = "none";
}
Appendix F

The Controller file

F.1 Controller actions)

This is a listing of mediator python functions defined in the C of our web2py powered prototype created in Chapter 6. It makes use of an enhanced controller approach recommended in Section 4.3.3, firing callable server-side functions. These firings define the application workflow and enable management of high-level application-specific spatial functions.

```python
def index():
    p1 = P('Identifier:', INPUT(_id='cidx', _value='None', _title='species identifier'))
    p2 = P('Name (English):', INPUT(_id='eng_name', _type='text', _value='None', _title='Species Name (English version)'))
    p3 = P('Name (Scientific):', INPUT(_id='listname', _type='text', _value='None', _title='Species Name (Scientific version)'))
    p4 = P('Name (Portuguese):', INPUT(_id='port_name', _type='text', _value='None', _title='Species Name (Portuguese version)'))
    p5 = P('Authority:', INPUT(_id='authority', _type='text', _value='None', _title='Legal responsible body'))
    p6 = P('Taxonomic order:', INPUT(_id='taxorder', _type='text', _value='None', _title='An order in a Biological taxonomy'))

    upform = FORM(INPUT(_name='BROWSE'))
    for i in q.executesql('SELECT cidx FROM distribution ORDER BY cidx'):
        OPTION(i[0], _value=str(i[0]))

    seled = OPTION('none', _value='none', _selected="selected")

    return dict(message='CONSISTENT WFS-T APP',
                 upform=upform,
                 p1=p1,
                 p2=p2,
                 p3=p3,
                 p4=p4,
                 p5=p5,
                 p6=p6,
                 seled=seled)

def update():
    uniqueIdentifier = request.vars['uniqueIdentifier']
```

149
cidx = request.vars['cidxo']
eng_name = request.vars['engo']
listname = request.vars['listo']
port_name = request.vars['porto']
authority = request.vars['autho']
taxorder = request.vars['taxo']
geometria = request.vars['geometria']
state = request.vars['state']

fid=''
for idfier in range(uniqueIdentifier.index('.')+1, len(uniqueIdentifier)):
    fid+=uniqueIdentifier[idfier]

if state == 'Update':
    try:
        triFunc = "update_distrib(%s, %s, '%s')" % (fid, cidx, geometria)
        q.fire(triFunc)
        session.flash="OK! update transaction of FEATURE %s commited successfully" % fid
    except:
        session.flash="Sorry!!! unable to update FEATURE %s" % fid

eelif state == 'Insert':
    try:
        triFunc = "add_distrib(%s, '%s')" % (cidx, geometria)
        q.fire(triFunc)
        session.flash = "INSERT transaction of FEATURE %s commited successfully" % cidx
    except:
        session.flash="Sorry!!! invalid geometry %s" % cidx

    try:
        triFunc = "add_species(%s, '%s', '%s', '%s', '%s', %s)" %
        (cidx, eng_name, listname, port_name, authority, taxorder)
        q.fire(triFunc)
        session.flash="OK! INSERT transaction of FEATURE %s commited successfully" % cidx
    except:
        session.flash="Sorry!!! unable to Insert FEATURE %s" % cidx

if state == "Delete":
    try:
        triFunc = "delete_distrib(%s)" % (uniqueIdentifier)
        q.fire(triFunc)
        session.flash="OK! DELETE transaction of FEATURE %s commited successfully" % fid
    except:
session.flash="Sorry!!! unable to delete FEATURE %s" % fid

else:
    session.flash="Unsupported action!!"

    redirect(URL(r=request,f="index"))

def restrict():
    uniqueIdentifier = request.vars["resIdentifier"]
    geom = request.vars["resGeometria"]

    geometria = textGeom(uniqueIdentifier, splitter(geom))

    try:
        hab = q.fire(geometria)
        session.flash="%s" % hab

        if hab=="True" or True:
            session.flash="GEOMETRY %s restricted to the specified list of interfluves" %
            uniqueIdentifier
    except:
        if uniqueIdentifier == "None" or None or "none":
            session.flash="You should first pick to-be restricted species distribution"
        else:
            session.flash="Unable to restrict GEOMETRY %s to the specified list of interfluves" %
            uniqueIdentifier
    redirect(URL(r=request,f="index"))

def addFam():

    fIdentifier = request.vars["fidx"]
    famname = request.vars["famname"]
    fTaxorder = request.vars["fTaxorder"]

    try:
        triFunc = "add_family(%s, %s, "%s")" % (fIdentifier, famname, fTaxorder)
        q.fire(triFunc)
        session.flash="OK! INSERT transaction of FEATURE %s commited successfully" % fIdentifier
    except:
        session.flash="Sorry!!! unable to Insert FEATURE %s" % \ fIdentifier
    redirect(URL(r=request,f="index"))

def addGen():

    gIdentifier = request.vars["gidx"]
    gennname = request.vars["gennname"]
gTaxorder = request.vars["gTaxorder"]

try:
    triFunc = "add_genus(%s, %s, "%s")" % (gIdentifier, genname, \
gTaxorder)
    q.fire(triFunc)
    session.flash="OK! INSERT transaction of FEATURE %s commited successfully" % gIdentifier
except:
    session.flash="Sorry!!! unable to Insert FEATURE %s" % \
gIdentifier
    redirect(URL(r=request,f="index"))

def shp2pg():
    import os
    dath = request.vars["fiload"]
    shp2pg = "/%s/static/PgTools/Tools/shp2pgsql.exe" % \
    request.application
    psql = "/%s/static/PgTools/Tools/psql.exe" % request.application
    dest = "/%s/static/web2pg.sql" % request.application

    SQLer="%s -g distrib -s 4326 %s -a distribution > %s" %\n    (shp2pg, dath, dest)
    toPG="%s -h 127.0.0.1 -d habtom -f %s -U postgres" % \n    (psql, dest)

    try:
        os.system(SQLer)
        os.system(toPG)
        session.flash="OK!!! shapefile [%s] inserted into database" %\n        dath
    except:
        session.flash="Sorry!!! unable to upload shapefile: [%s] into database" % dath
        redirect(URL(r=request,f="index"))

def splitter(geometria):
    splitted=geometria.split("/")
    splitted.remove(splitted[-1])
    return splitted

def textGeom(fid, splitted):
    qq="""restric_distrib(%s "") % fid
    for s in splitted:
        qq += ", " + """s"" % s
    return qq + ")""