Spring 5-18-2018

A Jython-based RESTful Web Service API for Python Code Reflection

John A. Nielson
University of New Orleans, jnielson@uno.edu

A Jython-based RESTful Web Service API for Python Code Reflection

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfilment of the
Requirements for the degree of

Master of Science
In
Computer Science

by

John A. Nielson

B.S., Clemson University, 2007

May, 2018

Table of Contents

Table of Figures

Abstract

Often times groups of domain experts, such as scientists and engineers, will develop their own software modules for specialized computational tasks. When these users determine there is a need to integrate the data and computations used in their specialized components with an enterprise data management system, interoperability between the enterprise system and the specialized components rather than re-implementation allows for faster implementation and more flexible change management by shifting the onus of changes to the scientific components to the subject matter experts rather than the enterprise information technology team. The Jython-based RESTful web service API was developed to leverage code reflection to provide access to Python scripts via RESTful web service calls, providing access to any function available in a Python script accessible to the REST server.

Keywords: Python; REST; reflection; interoperability; Jython

1. Introduction

Enterprise software systems typically contain scientific or knowledge-based components that are developed by domain experts such as scientists, business analysts who are not IT professionals. Their deep understanding of the specific process in the domain is often critical to the success of the entire business process and used by large number of end users in the related areas.

Integration of the scientific or knowledge-based components into the overall software system has been an active research topic with a number of approaches proposed such as End-User Programming (EUP) [Cypher et al. 2010], Web Mashup [Hoang et al. 2010], Scientific Workflow Management [Goecks et al. 2010], and End-User Process Modeling [Mukherjee et al. 2010]. The common idea adopted in these approaches is to provide the domain experts with some easy-to-use, half-done elements in order to reduce the efforts for the domain experts to make the elements in integration. Two essential requirements are inevitable: (1) The domain experts have to learn a software tool; the extent of success often depends on the learning depth. (2) Development and especially deployment need assistance of IT staff; as an element in the integrated software system, these activities are required to go through a software development process. Asking domain experts to study programming tools is never an efficient effort. Going through lengthy software process makes updates of scientific or knowledge-based components sluggish.

Rather than the software integration approaches, this thesis reports experiments of an approach to deployment of scientific or domain-specific components based on interpretations. In doing so, the domain experts use their own preferred programming language to implement the scientific or knowledge-based functionalities. The only information the client needs is the

module name of the method to be invoked; the protocol provides a list of all methods within that module as well as the names of each of those methods' parameters. The client can then invoke the method by providing its module, method, and parameter names according to the web service protocol. To publish the services of the scientific or knowledge-based components, RESTful Web service wrappers are automatically generated according to program name and the module, method, and parameter names by software reflection technique. Compared to the software integration approach, the system interoperation approach allows the scientific or knowledge components to be developed in an independent system out of the scope of the IT team managing the Web services. Deployment of new services and modification of existing services are both automated without delay of manual processing.

The system interoperation approach to deployment of scientific components has been applied to an initiative of a government project, the Probabilistic Reserves system for use in federal oil and gas regulation.

The goal of the Probabilistic Reserves project is to replace the legacy, deterministic calculation program with an online statistical tool that allows the user to choose a particular statistical distribution and enter multiple parameters and generate multiple percentiles in addition to the mean. These statistically generated ranges of values will assist analysts to account for the uncertainty associated with the underlying oil and gas reserves estimation. These reserve estimates are the basis for National Resource Assessments and the foundation for the United States' energy plan and policy.

2. System Background and Problem Statement

2.1 The Legacy Systems

The Probabilistic Reserves project was originally proposed in 2012 as an Oracle Forms and Reports-based solution that would incorporate newly developed probabilistic models into the reserve analysis process. Previously, the Oracle Forms and Reports-based tool, comprised of two different forms, only provided the capability of either manual entry or simple deterministic calculations of various attributes of a particular oil or gas reservoir, such as porosity, bulk volume, subsea contact depth, etc. In each case, a single value for these attributes was stored in the TIMS Oracle database.

When the project was originally conceived the vision was for it to be implemented via functionality added to the existing TIMS components. However, in the time between the project's inception in 2012 and its formal approval in 2016, the enterprise IT environment had begun its transition from a monolithic Oracle Forms and Reports-based application (TIMS) to a suite of SOA-based Java web applications (NCIS/TIMS Web) written in a variety of frameworks (pure Java EE, Oracle Web ADF, Vaadin 7 and 8). Because of this, a decision was made to replace the Reserves components of TIMS with a newly designed NCIS/TIMS Web application.

While both the legacy TIMS and modern NCIS and TIMS Web applications are both driven by Oracle databases, they are not the same instances, with the primary difference being that NCIS is a consolidation of the four TIMS regional databases (see section *Legacy Architecture*). All legacy reserve and reserve reservoir data will be migrated from TIMS to NCIS. However, there are other database objects not directly related to reserve calculations that consume this migrated data via views or package references. To preserve this functionality, data replication via Oracle Data Integrator (ODI) is used to ensure data is kept in sync between TIMS and NCIS.

2.1.1 Legacy System – TIMS Reserves Component

The legacy system used for performing reserves calculations is a component of TIMS, an Oracle Forms & Reports based application with and Oracle Database backend that uses PL/SQL packages for its calculations. Oracle Forms and Reports applications are Java applet based and require users to download Java code to be run on a client JVM.

Users log into one of the four TIMS databases, each one containing data from a different geographic region of the United States. In order to query or otherwise work with data from other regions, the user is required to log out of the system and log back into the regional database that contains the data the user is interested in for analysis. Below is a screenshot of the legacy TIMS Oil and Gas screen, one of the screens re-designed for the Probabilistic Reserves application:

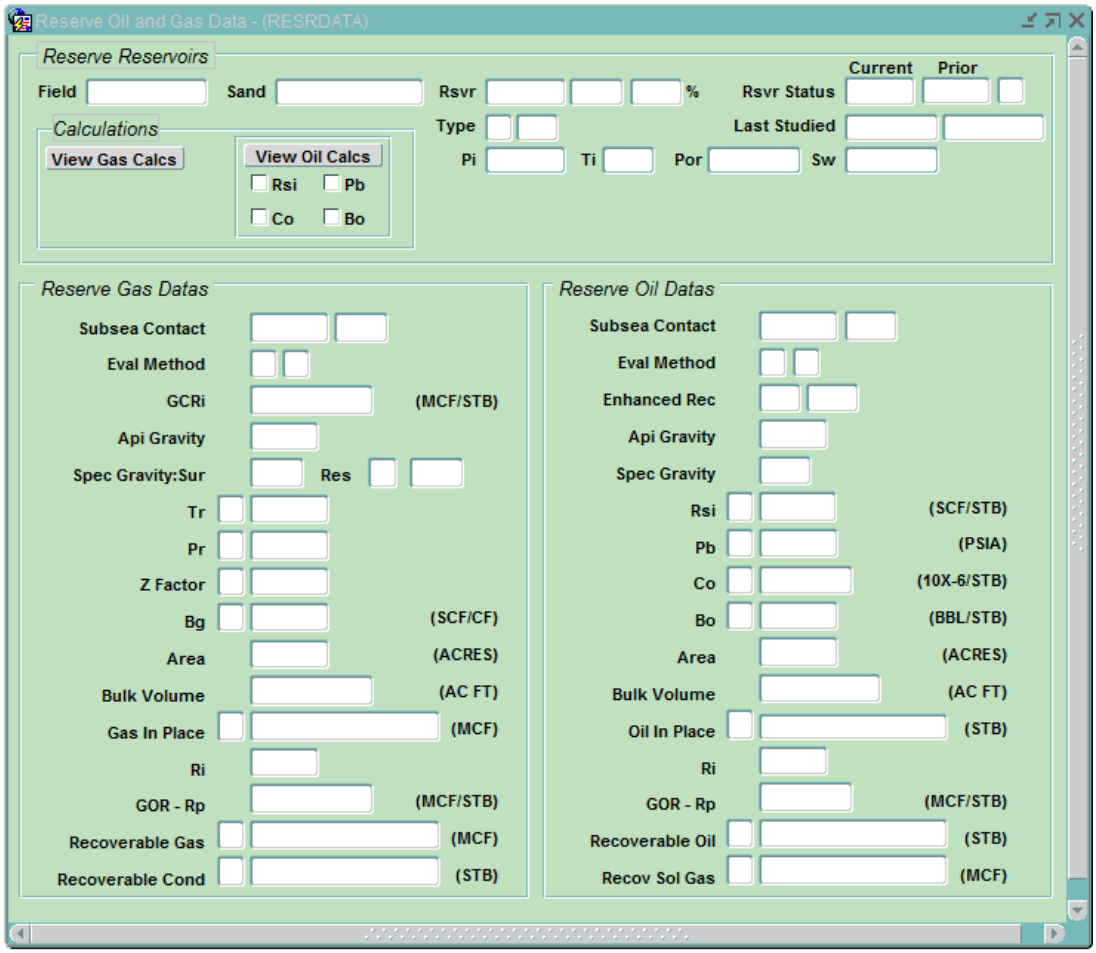

Figure 2-1: TIMS Oil and Gas screen (legacy application)

This screen allows the user to manually enter values for reserve reservoir attributes such as API gravity, specific gravity, subsea contact depth, area, bulk volume, etc. Some of these attributes are calculated within TIMS based on some parameters entered by the user. These calculations are performed by PL/SQL packages. For all of these attributes, i.e. those entered manually as well as those calculated in the application via PL/SQL packages, a single value is stored in the database.

2.2 The New System

The goal of the Probabilistic Reserves project is to replace the manual entry and deterministic calculations/formulae with a tool that will allow the user to choose a particular statistical distribution and enter a number of parameters and generate the 90th percentile (P90), 50th percentile/median (P50), 10th percentile (P10), and mean values for the selected distribution and parameters. In this case, four values – the P90, P50, P10, and mean – would be stored for each of these attributes, and the reservoir could be described by its probable characteristics. These estimates could then be used as input parameters for Monte Carlo simulations for forecasting; these simulations would also be included in the scientific code package provided by the subject matter experts. These reserve estimates are the basis for National Resource Assessments and the foundation for the United States' energy plan and policy, and these statistically generated ranges of values would allow analysts to better account for the uncertainty associated with oil and gas reserves estimation.

3. Related Work and Technical Background

A number of different technologies were used for the implementation of the Probabilistic Reserves system. These tools include server technologies, IDE plug-ins, various programming languages, libraries, and frameworks that made the integration of the TIMS Web application with the SME-provided probabilistic model code (Python modules) possible.

3.1 Server Technologies

Two different server technologies were used for the implementation:

- Oracle WebLogic 12c
- Apache Tomcat *7.0.59*

WebLogic was used for the Probabilistic Reserves application as accessed via the TIMS Web menu. WebLogic 12c was chosen because it is the organizational standard server technology for the enterprise environment, with all modern TIMS Web applications served via their own WebLogic instance in a VMWare ESXi virtualized server environment.

Apache Tomcat was used as the server platform for the web services that provide access to the probabilistic models. This server technology was chosen because it provides an easy implementation of Python-based servlets via the Jython library. The Python-based servlets, actually developed using Jython which provided access to the Java servlet libraries, allowed native access to the SME-provided probabilistic models via direct calls to their containing Python modules.

3.2 Development Frameworks

The Probabilistic Reserves system is comprised of two main components: the web application and the probabilistic model web services. The frameworks used for the development of these individual components are, respectively:

- Vaadin 8
- Jython Servlets

Vaadin 8 was chosen for the web application component of the system because it is an entirely Java-based development framework which does not require any separate HTML, JavaScript, or CSS to build a dynamic web application. Also, the framework's Designer plug-in for Eclipse, the IDE used in the enterprise development environment, allows for drag-and-drop design and implementation of web components which facilitate rapid application development.

For the web services component, Jython-based servlets served via Apache Tomcat were used. The Jython-based servlet method was chosen because it provided a well-known web

service endpoint that also allowed native access to the SME-provided Python modules containing the statistical distributions used to perform calculations within the system.

3.3 Development Tools

Different tools were used for the development of different components of the system. For the web application (Java/Vaadin) component as well as the web services (Python/Jython) components, Eclipse Neon was used. This choice was made for the following reasons:

- Vaadin 8 Designer, a WYSIWYG Vaadin application plug-in for Eclipse
- PyDev, an Eclipse plugin for Python development
- Easy WebLogic domain setup with existing enterprise test domain

Eclipse provides development tools for both main components of the system and therefore provided a centralized development environment particularly well-suited for this project.

For data modeling, Oracle SQL Developer was used. This tool is free and provides easy to use tools for data modeling/ERD generation that incorporates Oracle-specific features (data types, constraint types, etc.) as well as the generation of scripts used to build the physical data model in the database itself.

3.4 Database

The database technology providing access to the reserve reservoir data is Oracle Database 12c. The data model for the Probabilistic Reserves system was implemented in NCIS, the enterprise relational database used by various regulatory agencies for their analysis and other activities.

As part of the implementation, data was migrated to NCIS from four separate TIMS database instances, also Oracle Database. This migration was performed via SQL scripting. In

addition, in order to preserve the functionality of some other enterprise applications that consume the legacy TIMS reserve reservoir data that was migrated to NCIS, Oracle Data Integrator (ODI) is in use to replicate data between the two environments.

4. The Probabilistic Reserves System: Design, Development, and Key Technology

4.1Development Process

The Probabilistic Reserves system was developed over a series of distinct phases. These phases were:

- Initiation
- Discovery
- Development
- Transition
- Operations and Maintenance

These phases allow a mixture of Agile principles of adaptive requirements gathering while also facilitating conformance with enterprise governance principles and project management standard operating procedures within the organization. By breaking the project's development into those four stages, the project team was able to start the implementation with a well-defined set of requirements while still providing the SMEs with frequent opportunities to test and see demonstrations of the system as it evolved, finally culminating in a finished product that grew organically over the course of its implementation.

4.1.1 Initiation

The inception phase is when the highest-level planning of a new system occurs. This phase starts with a representative of the subject matter experts, typically the eventual end users of a system, preparing a document called a Mission Needs Statement that describes in general terms what the proposed system will do, an explanation of the need for the system, a description of some alternatives to the proposed system, and a description of the impact of not moving forward with the proposal. This document is what is presented to IT initiative decision-makers on whether or not to approve beginning the process of gathering further information regarding the development of a system.

If a Mission Needs Statement has been approved for further analysis, the assigned project manager will work with the subject matter experts to gather high-level requirements and build a tentative schedule and resource estimate for the development of the system. The estimated schedule and cost for the project are again presented for approval. If approved, the project moves to the Discovery phase.

4.1.2 Discovery

The discovery phase has two main outputs: the data model and the use cases. Both of these are produced through requirements elaboration sessions held with the subject matter experts. During these meetings the subject matter experts are interviewed to determine the workflows and business rules necessary to implement the processes required to reach the systems desired outcomes. Requirements are documented in a standardized enterprise use case template format.

The use case exists to document the steps and business rules required to achieve a particular outcome, i.e. the Success Condition Standard workflow is described by the Main Success Scenario. Any diversions/special cases that can be potentially encountered during the Main Success Scenario are captured as Scenario Extensions. Specific rules or other logic for the

various steps such as formulae or input validations are contained in the Business Rules section of the document. While the use case does not serve the purpose of a design document, the Design Considerations section contains suggestions, mockups, and/or design notes from discussions as well as screenshots of existing/legacy user interfaces on which the system is based.

4.1.3 Construction

The construction phase is the phase of the development process in which the technical implementation occurs. The construction phase was managed using an Agile Scrum-based methodology that facilitates iterative development with regular user testing periods – User Acceptance Testing (UAT) - throughout the project, along with design and requirements sessions to continue the elaboration and clarification of the project requirements.

Before the first iteration of development, each use case defined in the Discovery phase is broken up into discrete work items, or Product Backlog Items (PBIs). Each PBI represents a deliverable piece of functionality that is assigned to a particular developer.

Sprint Planning

As an iterative software development process, the construction phase is divided into six 4-week iterations, or Sprints. Each Sprint begins with a Sprint Planning meeting in which the development team (IT personnel) meets with the SMEs to prioritize all PBIs that have not been completed in a previous sprint. The goal is to decide on a subset of PBIs that will be delivered to and tested by the SMEs at the end of the Sprint during the Sprint UAT period. After the Sprint Planning meeting the development team begins work on the technical implementation of the PBIs chosen for that particular sprint.

Sprint Development

After the completion of the technical implementation of the PBIs chosen for the Sprint, the application is deployed to the Build and Test (BAT) environment for testing by the development team. This internal testing is performed by the development team's assigned tester via manual as well as automated testing using the Selenium automated web application test tool. Any bugs or issues uncovered during this testing period are reported to the development team and resolved. Once all bugs found in BAT have been verified and resolved by the tester, the application is then deployed to the Staging environment.

Sprint Testing

Once deployed to Staging, the application undergoes another round of the same manual and automated tests performed in BAT. Once again, any bugs or issues uncovered during this testing period are reported to the development team and resolved by applying the fixes first in the BAT environment, retesting the application in BAT as per the BAT internal testing process, redeploying to Staging, and tested internally once again.

Sprint UAT

Once all bugs found in Staging have been verified as resolved by the tester, the functionality developed in the preceding Sprint is demoed to the SMEs and made available to them for testing as part of the Sprint UAT. Just as the development team's tester reports bugs during his/her testing periods in BAT and Staging, all bugs uncovered by the SMEs during UAT are reported to the development team for verification and resolution. In addition to bug reporting, the SMEs also report any comments, concerns, clarifications on functionality, business rules, process flows, etc. All bugs and other feedback reported during this phase is tracked using a Sharepoint repository to which the SMEs have access. At the end of the Sprint UAT period, the feedback

items reported by the SMEs are reviewed and discussed, with the development team explaining what steps were taken to resolve any issues as well as to create new PBIs for any new requirements identified or requested as part of UAT.

4.1.4 Transition

After completion of the final Sprint, the project moved into the Transition phase. During this period final steps are taken to verify the completeness of the application, i.e. whether all of the baselines requirements have been addressed and the project meets its high-level goals, and the application then enters its final UAT period, the Integrated UAT phase.

Integrated UAT is the final opportunity for the SMEs to test the functionality of the system in its entirety before it is released to Production. The version of the system tested during Integrated UAT includes all bug fixes discovered to date as well as any additional requirements requested and approved for development in preceding Sprint Planning meetings. After a successful Integrated UAT phase in which all bugs are fixed, the SMEs determine that the system addresses all of its high-level requirements, and the ATD and ATC are approved, the system as it has been developed is considered to be accepted by the SMEs and is ultimately released to the Production environment.

4.2 Challenges

The crux of the application, however, is the set of Python modules that contain the code for the various statistical distributions usable for the reserve reservoir data. This code was developed separately by the subject matter experts and is not intended to be maintained by information technology staff. Rather the ultimate goal of Probabilistic Reserves is to provide a web service framework and client application that can analyze the Python scripts themselves, provide access to their internal functions which produce the various statistical distributions to be used for geological and geophysical analysis. These are developed and managed independently from the

IT team managing the Web services of the Probabilistic Reserves project. If they are treated as components integrated into the system, then any change and update would have to go through the process described in Section 4.1, which will be lengthy and costly.

Furthermore, since the scientific components are developed in Python, but the web application is Java-based residing on WebLogic server, the application required the Jython PythonInterpreter object. This Java-native technique is troublesome in integration for several reasons.

1. `PythonInterpreter` works by invoking scripts stored on a filesystem and executing commands via string literals and its own Jython API, requiring cumbersome method and module calls from Java code
2. The web server on which the Probabilistic Reserves application resides, WebLogic, is bundled with its own distribution of Jython which is outdated and does not support the Python `inspect` module used for code reflection. Although bundled libraries can be overridden using the <prefer-application-packages> or <prefer-application-resources>, any Python errors thrown by the interpreter cause the application to revert to the bundled Jython library

Eventually, a separate web server was created to enable the Python code to be invoked by a web service to interoperate with the WebLogic-based application via RESTful web service calls.

4.3 Business Workflow

The Probabilistic Reserves system is composed of three major components:

- The Java web application server (WebLogic)

- The Jython servlet and scientific component (Python scripts) server (Tomcat)
- The database (Oracle)

As shown in the sequence diagram below, the client interacts directly with the web application via the WebLogic server and selects a scientific component (Step 1). He or she first opens a menu that sends a web service request to the Tomcat server for all methods available in the selected scientific component (2). The servlet on the Tomcat server then uses the inspect module on the specified component, actually the name of a Python module, and returns a JSON object of a map of all of its available functions and their parameters (3). The user then selects one of these functions, enters the parameters needed for the function (4), and then the WebLogic server makes another web service call to the Tomcat server with the specified function and parameters (5). The servlet then directly calls the specified method of the scientific component with the parameters provided, formats them as a JSON object, and returns it to the WebLogic server (6), which then formats and displays that output to the user in the web application (7). If the user chooses, he or she can store the output in the database (8, 9).

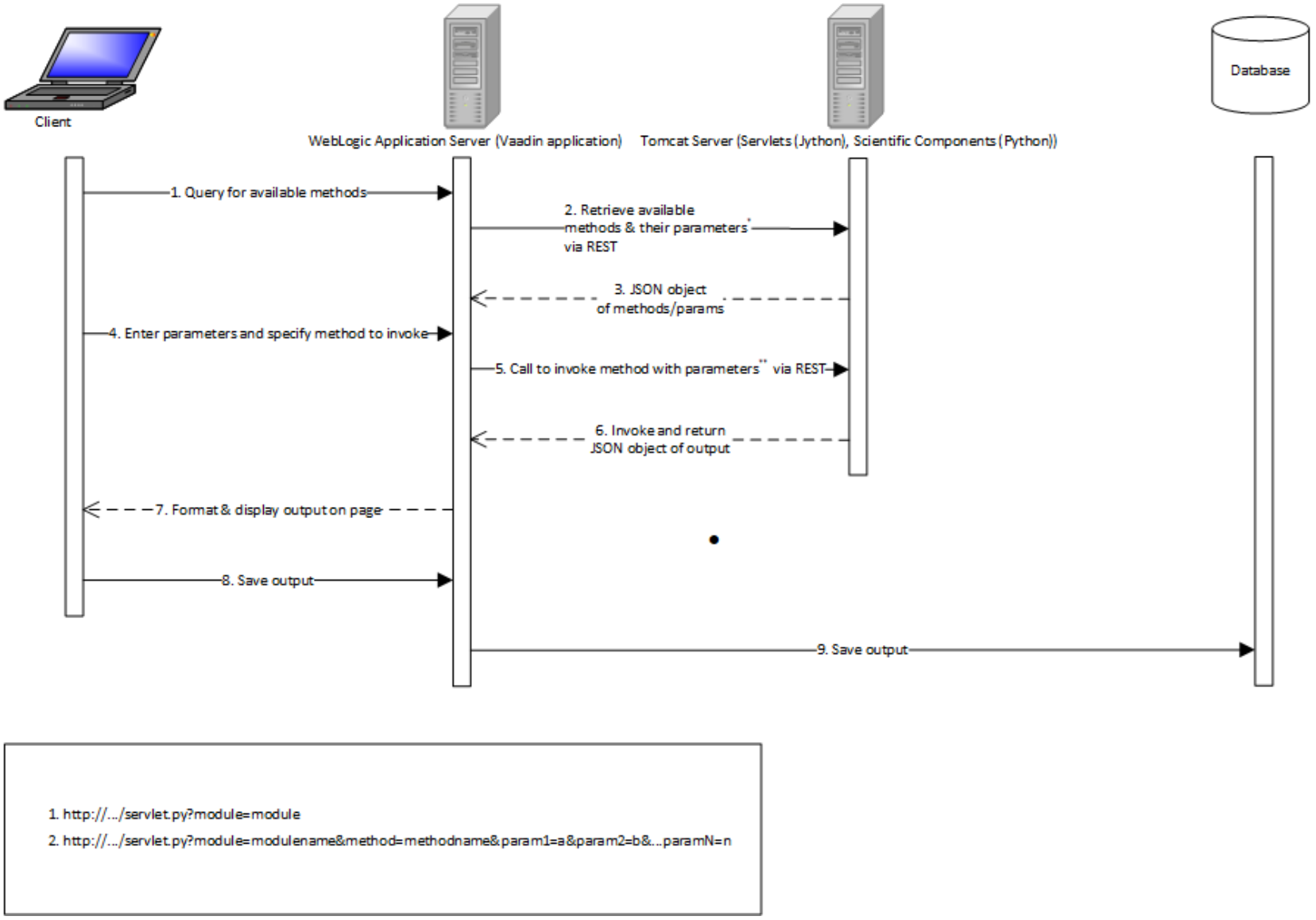

Figure 4-1: System architecture and sequence of requests for Probabilistic Reserves calculations via Jython web services

4.4 Data Model

The conceptual data model that shows the relationship between reservoirs and their oil and gas reserves is below:

Figure 4-2: Legacy conceptual data model. The data model above shows the relationship between reservoirs and their oil and gas reserves as defined by the reservoir's ID. The reservoirs as well as their reserves certain properties/attributes for which statistical

Because the system now captures four different values – P10, P50, P90, and Mean – rather than a single value for each particular attribute, the data model was expanded. Each column in the reservoir and oil and gas reserve tables for the various attributes was replaced with a foreign key reference to a separate table of rows referencing that particular execution of the selected distribution. Each row contains the name of the distribution used, the calculated values for P10, P50, P90, and Mean, as well as a foreign key reference to another table of reserve reservoir distribution parameters that track the name and value of each parameter used for the distribution. This allows the database to track executions of distributions and parameters with names that can

change at any time, facilitating the flexibility allowed by Python scripts analyzed via code reflection and accessed by web services. The users can alter these Python scripts at any time to modify the code for the distributions as well as add or remove distributions. This can be done by a simple re-deployment of the Python code without any changes to the application. Below is the updated data model that accommodates the changes described:

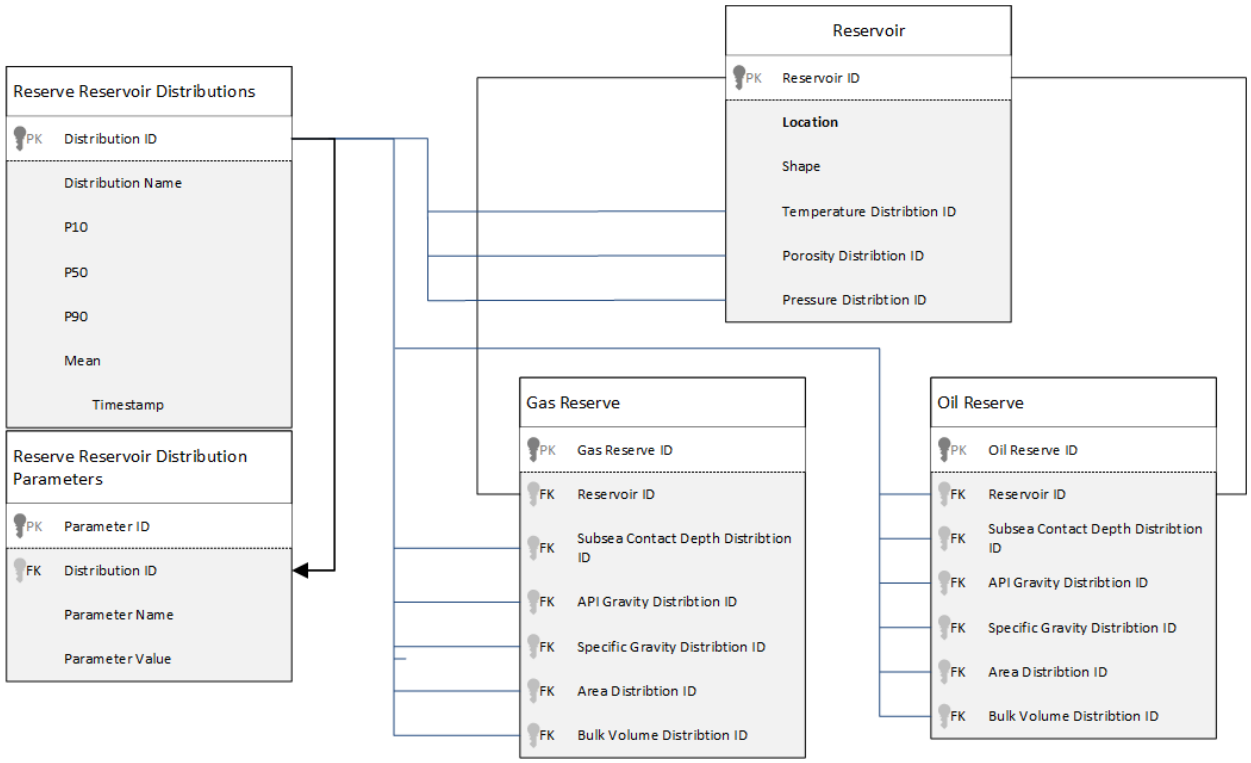

Figure 4-3: Updated data model containing new tables to track distributions and their parameters

4.5 Design – Jython Web Service Protocol for Code Reflection

A number of different options exist for providing Python web services. For example, the Flask framework allows Python code to be exposed natively via web services, and of course CGI is available as well. However, Apache Tomcat was chosen because the enterprise environment already provided support for that server platform.

In order to expose Python code as web services on a Tomcat server, a couple of changes are required. First, the Jython JAR must be deployed to the application's `/lib` directory (`$CATALINA_HOME/webapps/$APP_NAME/lib`). First, a servlet mapping entry in the application's web.xml must be created to instruct the server to route requests to a Python script rather than a WAR or other Java-based web application. [Juneau, Wierzbicki and Baker 2010]

```
<web-app>
	<servlet-name>PyServlet</servlet-name>
	<servlet-class>org.python.util.PyServlet</servlet-class>
	<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
	<servlet-name>PyServlet</servlet-name>
	<url-pattern>*.py</url-pattern>
</servlet-mapping>
</web-app>
```

The servlet code itself is straightforward and familiar. Jython provides access to the Java servlet libraries via Python code. Below is an example of a Jython servlet that calls a method, `get_functions`, that returns all functions defined in a specified module:

```
from javax.servlet.http import HttpServlet
import pr_util

class pr_servlet (HttpServlet):
	def doGet(self,request,response):
		self.doPost (request,response)

	def doPost(self,request,response):
		toClient = response.getWriter()
		response.setContentType ("application/json")


		map = request.getParameterMap()

		print "map = %s" % map

		module_name = map['module'][0]

		functions = pr_util.get_functions(module_name)


		toClient.print(functions)
```

This servlet returns a JSON object of all functions in the specified module mapped to their specified parameters. It does so by calling the function `get_functions`, which uses the Python inspect module for code reflection. The `get_functions` module is below:

```
import inspect
import json

# Returns a dict that maps function names to a list of its arguments
def get_functions(module_name):

	print "Gettin' functions"
	functions = {}

	module = __import__(module_name)

	members = inspect.getmembers(module)

	print "Got members"

	# The members of a module come back in this format:
	#	('module_name', <type module_name at 0x73>) or some other hex
	#	address
	for member in members:

		if inspect.isfunction(member[1]):

			# The function's name

			func = str(member[0])

			# Initialize the list of args
			functions[func.encode('utf-8')] = []
			argspec = inspect.getargspec(member[1])

			# Build the list of args
			for arg in argspec.args:

	functions[func.encode('ascii')].append(arg.encode('utf-8'))

	functions = json.dumps(functions)

	return functions
```

The function above returns a JSON object that represents a mapping of all functions defined in the specified module along with all parameters for each defined function. Below is an example

response to `get_functions`, built from a Python dictionary, containing two defined functions for the log gamma and mean of Weibull calculations, respectively:

```
{"lgam":["z"], "weilm":["a","b"]}
```

In this example, you have two different distributions available to you, each with a different number of parameters. The log gamma distribution, labeled "lgam," takes one parameter, z. The Mean of Weibull distribution, labeled "weilm," takes two parameters, a and b.

When the user first opens the calculation dialog, the web application requests the methods and their parameters in the module "beta1.py" from the Tomcat server via the Java API for REST services:

```
public static HashMap<String, ArrayList<String>> getMethods(String
moduleName) {

            HashMap<String, ArrayList<String>> methods;
            Client client = ClientBuilder.newClient();
            WebTarget target;
            String resp;
            JsonObject jo;
            JsonArray ja;

            target = client.target("http://localhost:8080/tw-
            pythonservices/pr_servlet.py").queryParam("module", "beta");

            resp =
                  target.request(MediaType.APPLICATION_JSON).get().readEntity
                  (String.class);

            jo = Json.parse(resp);

            System.out.println("jo = " + jo);
            /*
             * Each key of the JsonObject is the name of a method. The values
             * are JsonArrays of the names of the parameters for those
             * methods.
             * /
            methods = new HashMap<String, ArrayList<String>>();

            for (String k : jo.keys()) {

                  methods.put(k, new ArrayList<String>());

                  ja = jo.getArray(k);
```

```
                for (int i = 0; i < ja.length(); i++) {
                      methods.get(k).add(ja.getString(i));
                }
          }

          return methods;
    }
```

First, to create a distribution for an attribute such as subsea contact depth for an oil reserve, the user first selects the reservoir and reserve in the Oil and Gas data interface and clicks the button for the specified attribute, in this case “Subsea Contact” (highlighted):

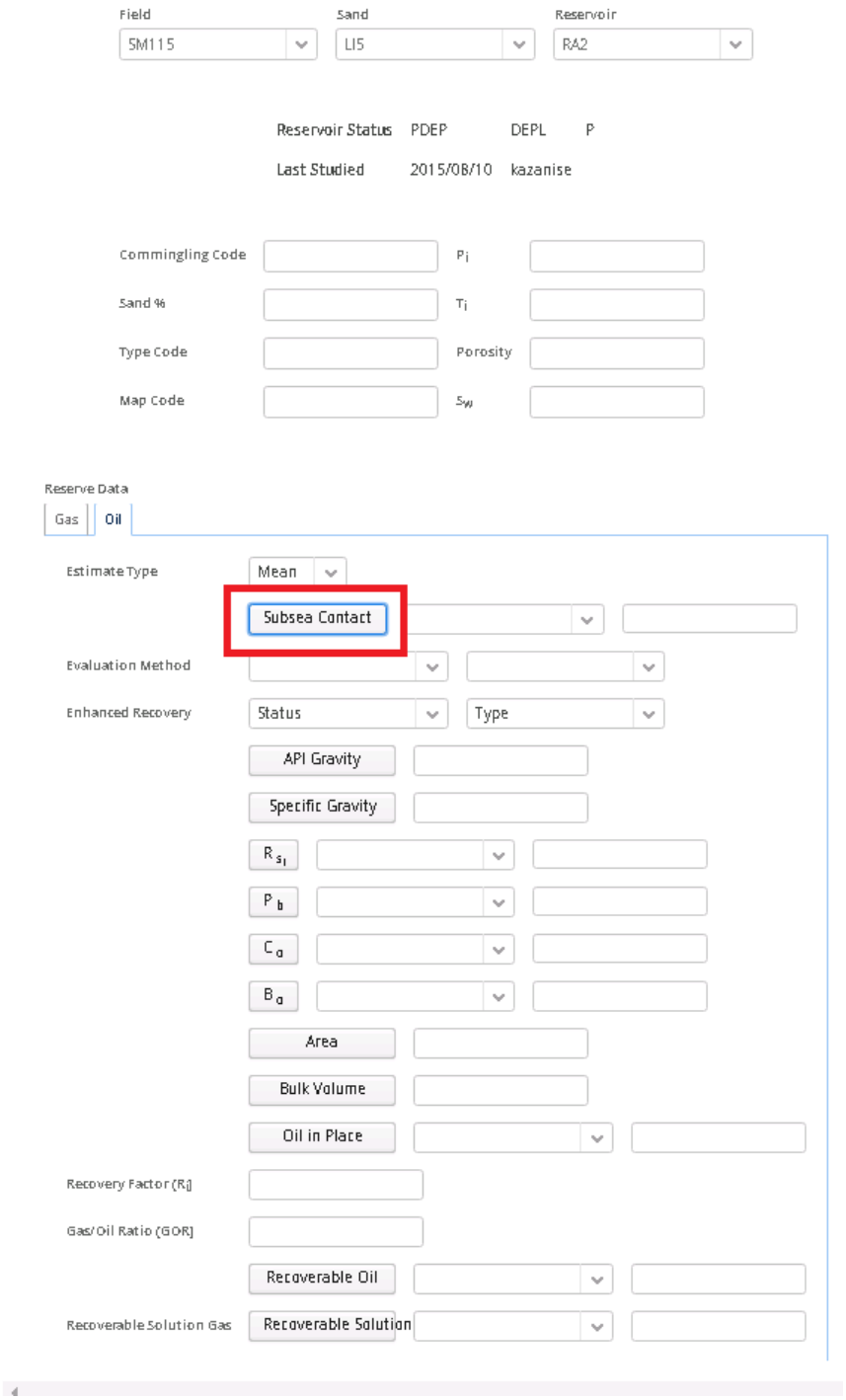

Figure 4-4: The Probabilistic Reserves Oil/Gas reserve data interface

The application then presents the Reserve Calculations Dialog which allows the user to select a distribution, enter its parameters, create the distribution, and store the results. Before a

distribution has been selected, there are no parameters available for input in the Calculation dialog (empty area highlighted):

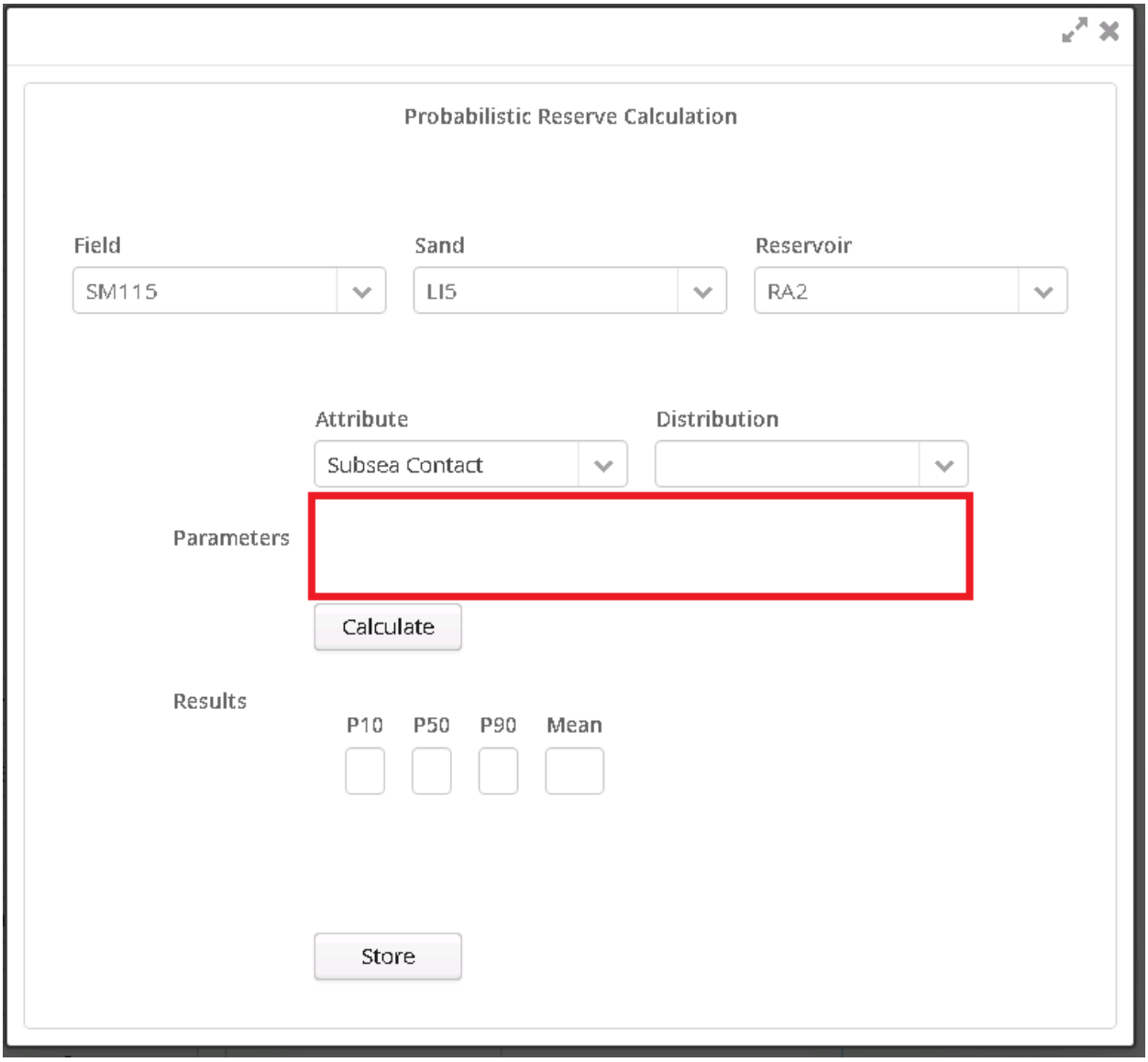

Figure 4-5: Reserve Calculation dialog. Because no distribution has been selected, no parameter input fields are displayed.

After selecting a distribution such as lgam, the Parameters section of the interface is updated with the parameter for the selected distribution:

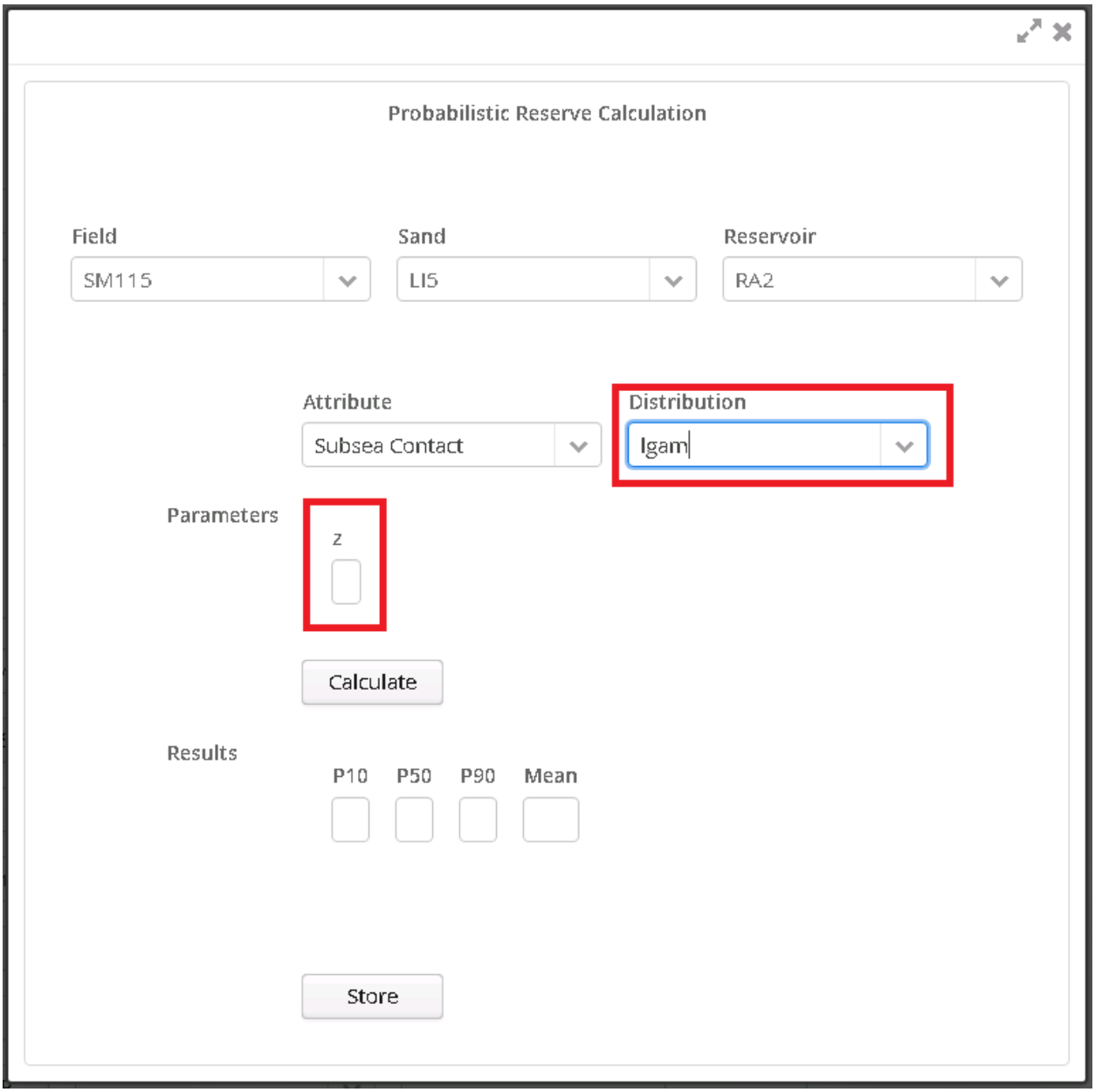

Figure 4-6: Reserve Calculation dialog. The lgam distribution has been selected, so its one parameter, 'z', is now available for input

If the user selects a different distribution, the Reserve Calculation dialog is updated with the newly selected distribution's parameters, in this case 'weilm':

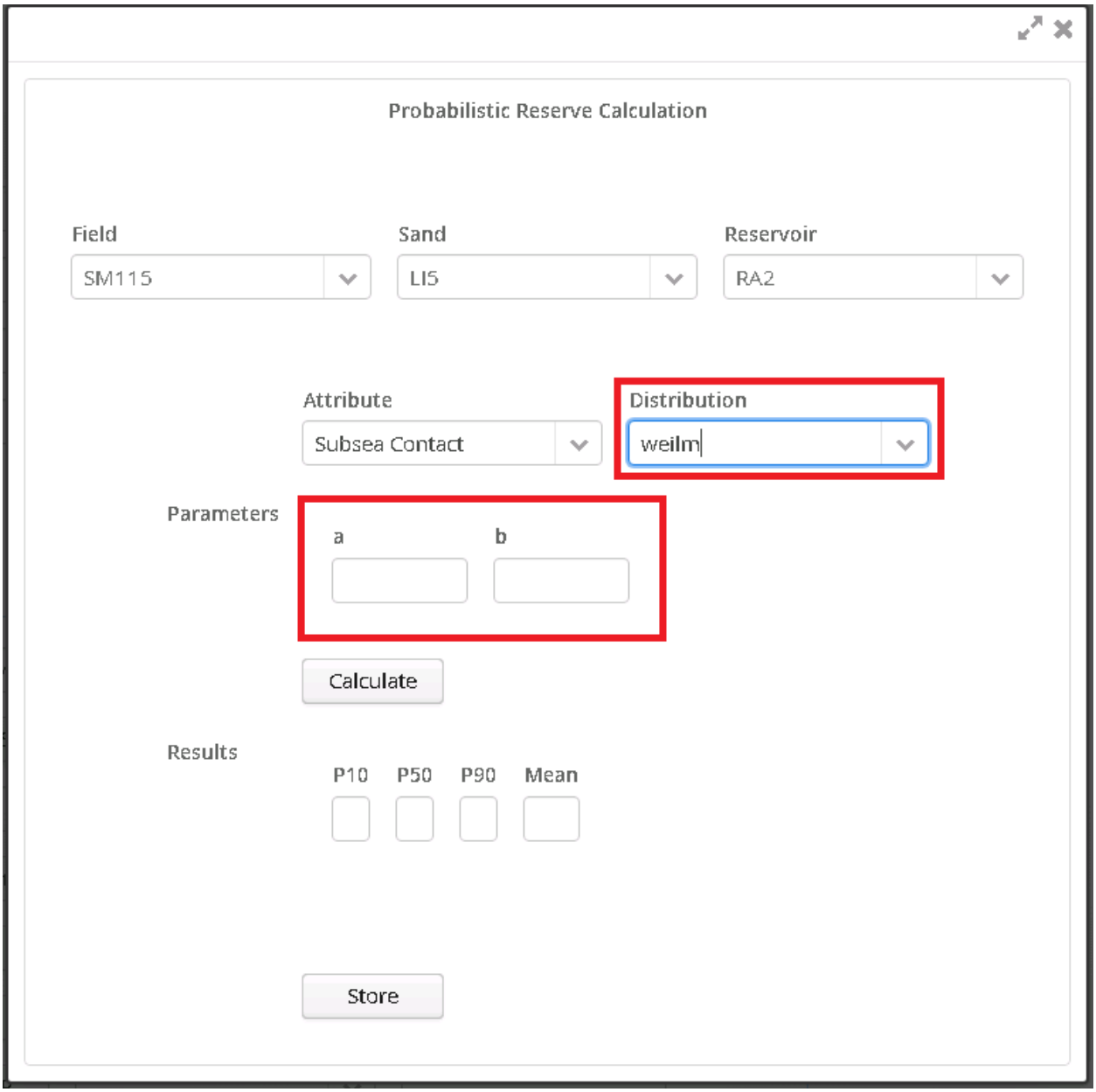

Figure 4-7: Reserve Calculation dialog. The selected distribution was changed to 'weilm,' so now its two parameters, 'a' and 'b,' are available for input

To generate the values for P10, P50, P90, and Mean for the selected attribute using the selected distribution, the user enters values for each parameter and clicks ‘Calculate.’ After clicking the ‘Calculate’ button, a new web service request is made by the web application to the Tomcat server specifying the method to invoke (in this case “weilm”) with the specified parameters.

Then, using the Python `inspect` module, the Jython web service invokes the specified method by name with the specified parameters. Below is the Java web application code, the function `PythonUtil.invokeMethod(String moduleName, String methodName, Map params)` that calls the Jython web service via the request http://localhost:8080/tw-pythonservices/pr_servlet.py?module=*beta1*&method=*weilm*&a=*1*&b=*2*:

```
public static JsonObject invokeMethod(String moduleName, String methodName, Map
params) {

		HashMap<String, ArrayList<String>> methods;
		Client client = ClientBuilder.newClient();
		WebTarget target;
		String resp;
		JsonObject jo;
		JsonArray ja;
		String url;

		url = "http://localhost:8080/tw-pythonservices/pr_servlet.py";

		// Build the initial WebTarget object with the method name specified
		target = client.target(url).queryParam("module", moduleName);
		target = target.queryParam("method", methodName);

		// Iterate over parameters to set queryParams of WebTarget
		for (Object param : params.keySet()) {

			target = target.queryParam(param.toString(), params.get(param));

		}

		resp =
target.request(MediaType.APPLICATION_JSON).get().readEntity(String.class);

		System.out.println("resp = " + resp);

		jo = Json.parse(resp);

		System.out.println("invokeMethod.jo = " + jo.toString());

		return jo;

	}
```

…which in turn calls the Jython servlet which calls the Python/Jython method invoke which uses the inspect module to invoke the function by name:

```
# Invokes the specified method (string) in the specified module (string) with the
specified parameters (dict: {string:string})
def invoke(module_name, method_name, param_map):

	module = __import__(module_name)

	method = getattr(module, method_name)

	args = inspect.getargspec(method).args

	print "param_map before: %s" % param_map

	# Remove any invalid arguments
	for key in param_map.keys():
		if key not in args:
			del param_map[key]

	print "param_map after: %s" % param_map

	output = method(**param_map)

	print "output = %s " % output

	return output
```

Here, the Python method `invoke` performs three major steps that utilize code reflection to call the specified function:

1. It imports the specified module via __import__(…)

   ```
   module = __import__(module_name)
   ```

2. It obtains a reference to the method via getattr(…)

   ```
   method = getattr(module, method_name)
   ```

3. It calls the method by passing a kwargs dictionary of the GET query parameters to the method's reference and returns the output

```
output = method(**param_map)
```

The protocol effectively translates the GET string

```
http://...?module=beta1&method=weilm&a=1&b=2
```

into the Python method call

```
beta1.weilm(a=1, b=2)
```

The output is returned to the application in the form of the values for P10, P50, P90, and Mean. To store these values in the database and associate them with the selected reservoir/reserve, the user clicks ‘Store.’

5 Future Considerations

5.1 GIS Integration

Data entered and stored via the TIMS Reserves components is also consumed by the TIMS GIS mapping system. This system uses ArcGIS Desktop to view spatial data related to the reserves and reservoirs maintained in TIMS. ArcGIS Desktop accesses the TIMS Oracle database via proprietary middleware, ArcSDE, which allows various GIS applications to interpret relational data as spatial data. For example, once a reservoir is saved in TIMS, that reservoir can be displayed on a map as a polygon-type spatial feature. This process is outlined in the figure below:

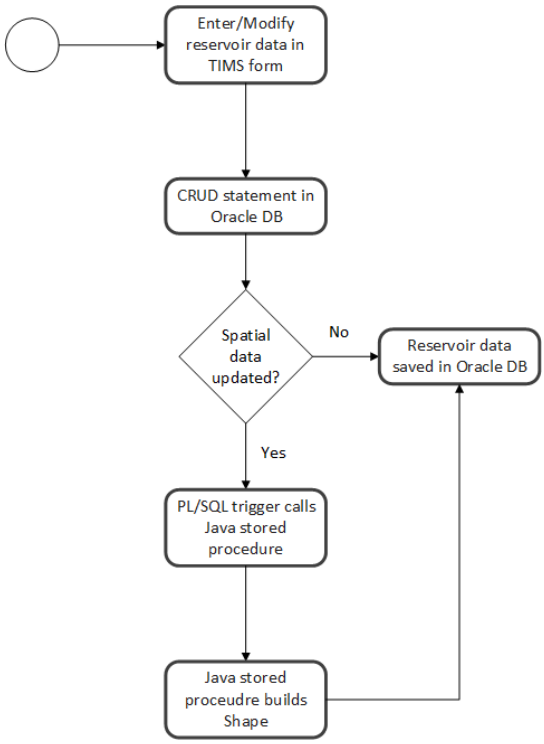

Figure 5-1: The spatial update process. When a record is updated/inserted/deleted in(to)/from tables with spatial data, a PL/SQL trigger is executed that initiates a Java stored procedure that creates the shape object according to application-specific business rules. This data, along with the rest of the row, is stored in the SHAPE column of the modified table, in this case Reservoirs

The spatial features are stored as spatial data in the TIMS database using the ST Geometry data type in the TIMS Oracle database. Each feature is represented by a single row in the database, and the attributes of the feature – its location, coordinate system, geographic boundaries, etc, are stored in a ST Geometry data type column called Shape.

Because rewriting and/or migrating the spatial update process from TIMS to NCIS was outside the scope of the project, a solution was needed to preserve the spatial updates while still using the new Probabilistic Reserves application as the entry point for reservoir data. In order for the PL/SQL triggers to fire and in turn call the Java stored procedures that build the spatial data (value for Shape) in the database, data replication was required. This was achieved via the Oracle Data Integrator (ODI) tool, for which mappings for corresponding tables across databases

are created. When DML for a table is executed in NCIS, a corresponding DML statement is executed in TIMS via ODI which keeps the data in sync. Since a DML statement is executed, the trigger still fires which results in a call to the Java stored procedure. The updated spatial update process for data updated in TIMS via ODI is shown below:

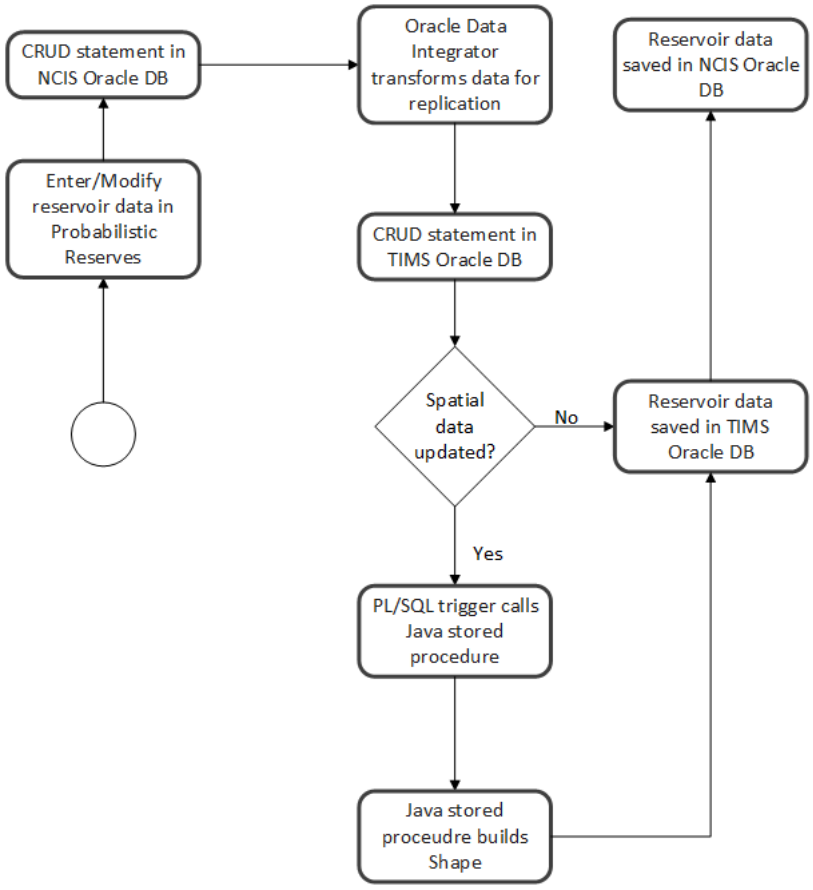

Figure 5-2: Updated spatial update process incorporating ODI for data replication

6. Conclusion

Ultimately, the choice to implement the web services using Jython served via Tomcat was largely due to necessity. In an enterprise environment, often the implementation is very much dictated by the technology stack currently in place. However, the technical requirements for the system, i.e. a dynamic system that would allow a web application to invoke mathematical functions written in Python and maintained by an external entity, were easily addressed by the technology available. Tomcat offers an easy way to serve Python-based servlets via the Jython library, and the Python inspect module, combined with other Python built-in code reflection modules such as `__import(…)__` and getattr(…), provided all the tools necessary to allow Python modules to be inspected to determine all of its available defined functions and their parameters, and the functions can be invoked by name with named parameters. All of these features comprise a tool that can instantly produce a web-based API providing function access for any given Python module with no modification necessary by the developer of the module.

References

CYPHER, A., DONTCHEVA, M., LAU, T., AND NICHOLS, J., Eds. 2010. No Code Required - Giving Users Tools to Transform the Web. Morgan Kaufmann.

GOECKS, J., NEKRUTENKO, A., TAYLOR, J., AND TEAM, T. 2010. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biol 11, 8, R86.

HOANG, D., H., P., AND NGU, A. 2010. Spreadsheet as a generic purpose mashup development environment. In ICSOC. 273–287.

JUNEAU, J., WIERZBICKI, F., BAKER, J., SOTO, L., NG, V. 2010. Chapter 13: Simple Web Applications - Jython Book v1.0 documentation

http://www.jython.org/jythonbook/en/1.0/SimpleWebApps.html

MUKHERJEE, D., DHOOLIA, P., SINHA, S., REMBERT, A., AND NANDA, M. 2010. From Informal Process Diagrams to Formal Process Models. In BPM'10. 145–161.

Vita

John Nielson was born in Fayetteville, NC, but spent most of his life before college in Watkinsville, GA. He received his B.S. in Mathematical Sciences with an emphasis in Applied and Computational Analysis from Clemson University in 2007. He then moved to New Orleans and began his career as a software engineer developing GIS applications. He currently works as a software development project manager with the Bureau of Safety and Environmental Enforcement.