**RESTful Spring Based Web Service: ANOVA Calculation**

Object Oriented Software Design

Keith Rodgers

05/10/2016

# Table of Contents

[Table of Contents 2](#_Toc450607781)

[Problem 3](#_Toc450607782)

[Background 3](#_Toc450607783)

[Task 3](#_Toc450607784)

[Methodology 3](#_Toc450607785)

[Architecture 3](#_Toc450607786)

[Implementation 4](#_Toc450607787)

[Execution 4](#_Toc450607788)

[Query 6](#_Toc450607789)

[Conclusion 8](#_Toc450607790)

[Result 8](#_Toc450607791)

[Tools 9](#_Toc450607792)

[Appendix 9](#_Toc450607793)

[Web Service Sample 9](#_Toc450607794)

[Form 10](#_Toc450607795)

[Confirmation 10](#_Toc450607796)

[Query 11](#_Toc450607797)

[Class Diagram 11](#_Toc450607798)

# Problem

## Background

Researchers performing scientific experiments typically use a series of statistical models to analyze the significance (or lack thereof) of their treatment on the intended subjects. One of these models is ANOVA, a method testing whether the means of several groups are equal. Whereas the t-test typically applies to 2 groups, this typically applies to 3 or more groups which are monitored.

## Task

For this project, I was assigned the task to build a REST-ful Spring web service in Java which performs this ANOVA calculation. The requirements were as follows:

* + **Execute**: Takes input data and returns the result when the computation is done
  + **Submit**: Returns an integer job id immediately without blocking the computation
  + **Query**: Takes input integer job id and returns the calculation or indicates the computation is not done

In addition, the desired requirements also included that this web service be in a RESTful state. The principle of a RESTful web service include:

* + **Resources** expose easily understood directory structure URIs.
  + **Representations** transfer JSON or XML to represent data objects and attributes.
  + **Messages** use HTTP methods explicitly (for example, GET, POST, PUT, and DELETE).
  + **Stateless** interactions store no client context on the server between requests. State dependencies limit and restrict scalability. The client holds session state.

# Methodology

## Architecture

I designed the program follow the MVC architectural structure. The controller managed all actions through HTTP calls. The pattern for the program is shown below:

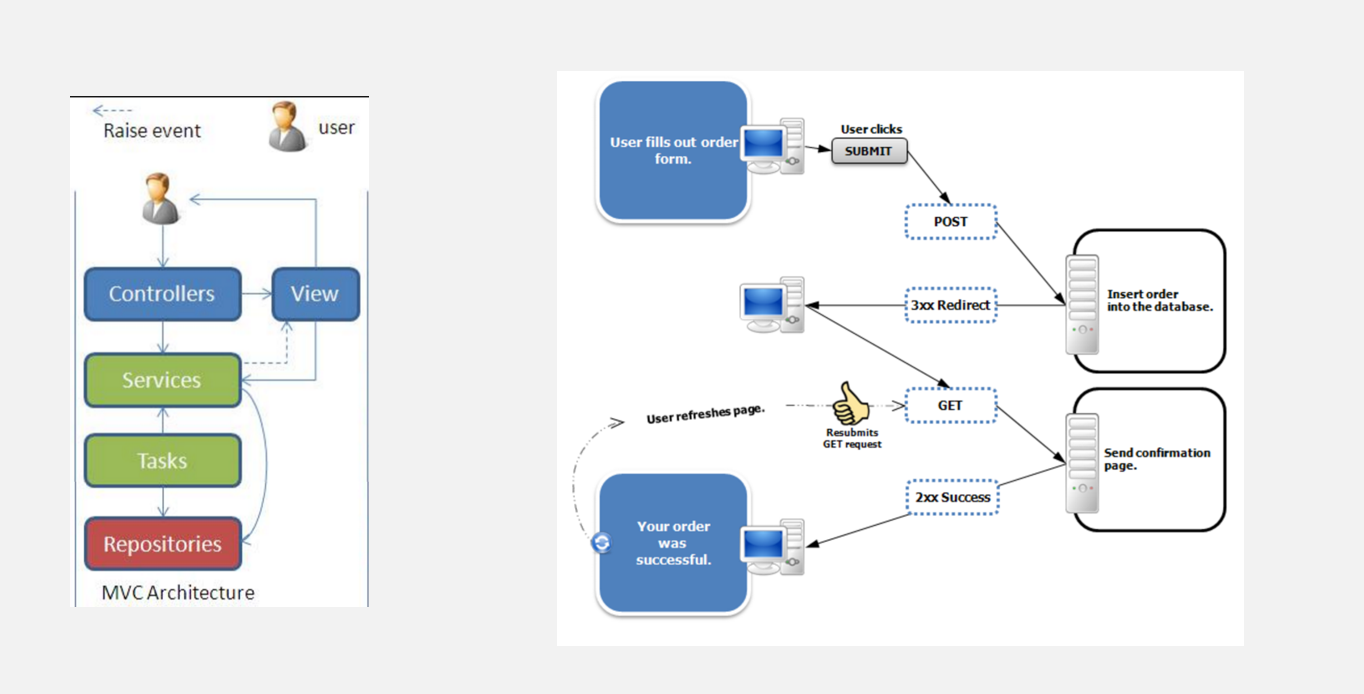


Figure 1: MVC-RESTful Architecture

## Implementation

### Execution

To implement the stated requirements, the program needed to collect data and pass it to a set of pre-built classes for execution. These pre-built classes performed the actual ANOVA calculation and were implemented by the geWorkbench project. To start, I implemented a web controller which managed the presentation, model, and any actions to perform the ANOVA calculation. Any time the browser client sent a HTTP GET request to the ‘/’ (shown below) resource, the ‘form’ (html page) was accessed and shown to the user. In addition, I binded the View to class objects to capture and process the data. See below:

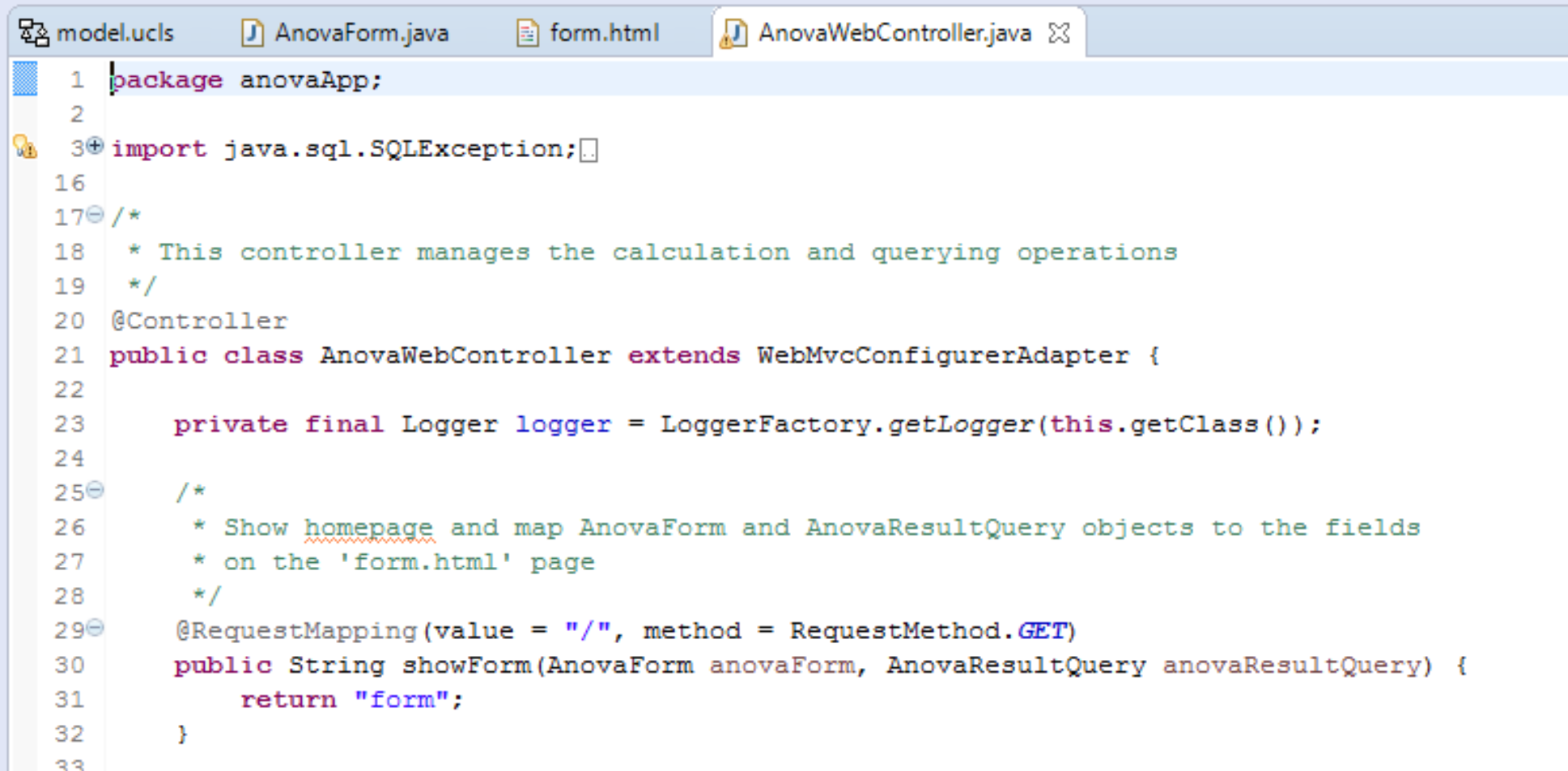


Figure 2: Web Controller

On the form page, when the ‘Submit’ button is selected, the fields from the form were binded to the AnovaForm object upon HTTP POST operation.



Figure 3: Mapping AnovaForm to the form View

I defined the ‘calculateAnova’ method, which was a HTTP POST operation. This controller method starts the ANOVA calculation process by setting all the values from the AnovaForm object to the AnovaInput object (AnovaInput from geWorkbench project).

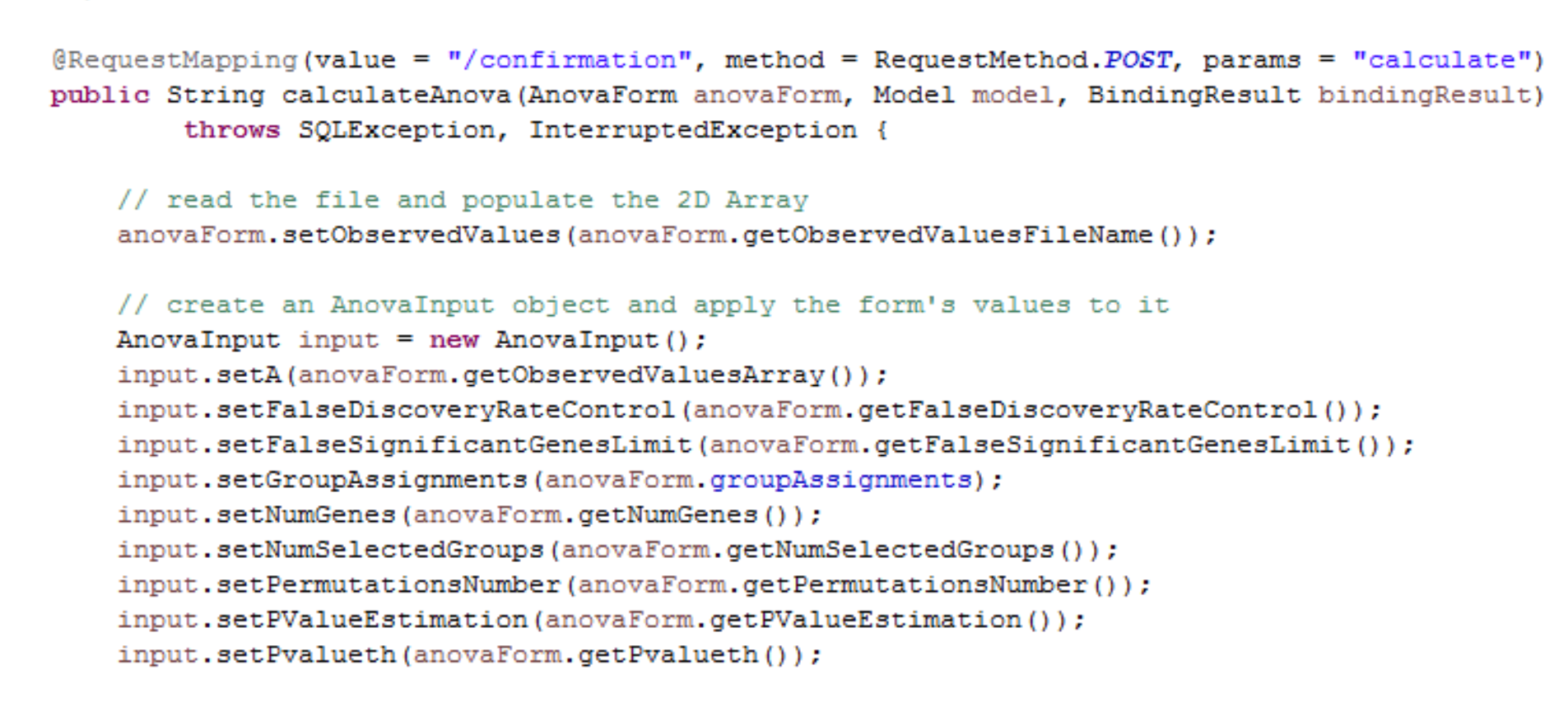


Figure 4: Web Controller HTTP Post to kick off computation

Once the AnovaInput class object was properly setup, the calculation process could begin. However, the program needed to meet the requirement that a job id is returned immediately without blocking the ANOVA computation. To do this, I created a simple database (using SQLite) with a single table, AnovaResultsTable, for data persistence. The AnovaDatabase class contains a method to establish a connection to the database as well as a method to create a new row in the database. So, in the AnovaWebController, I created an AnovaDatabase object to create a new row immediately in the database upon each HTTP POST call to the server.

Then, I created a class ‘MyRunnable’ which implements the Runnable interface so that the computation could run on a separate thread. Once the computation finishes, an UPDATE sql statement updates the database table using the last job id executed.

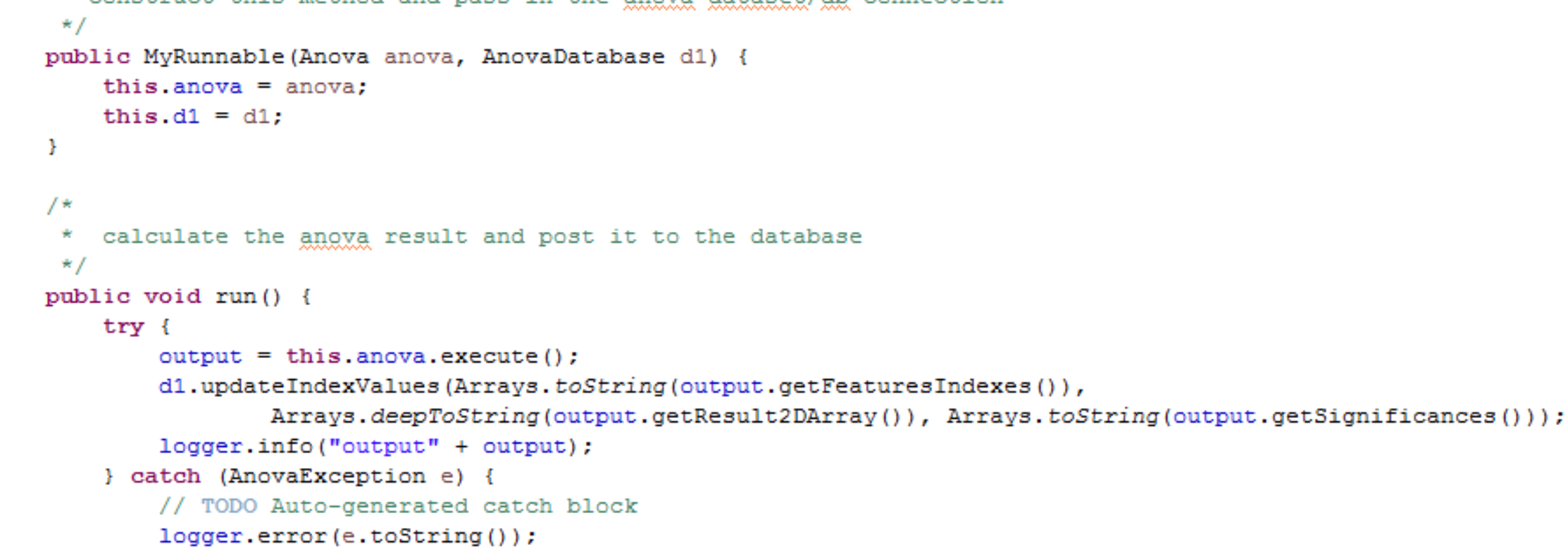


Figure 5: Implementation of the Interface method run()

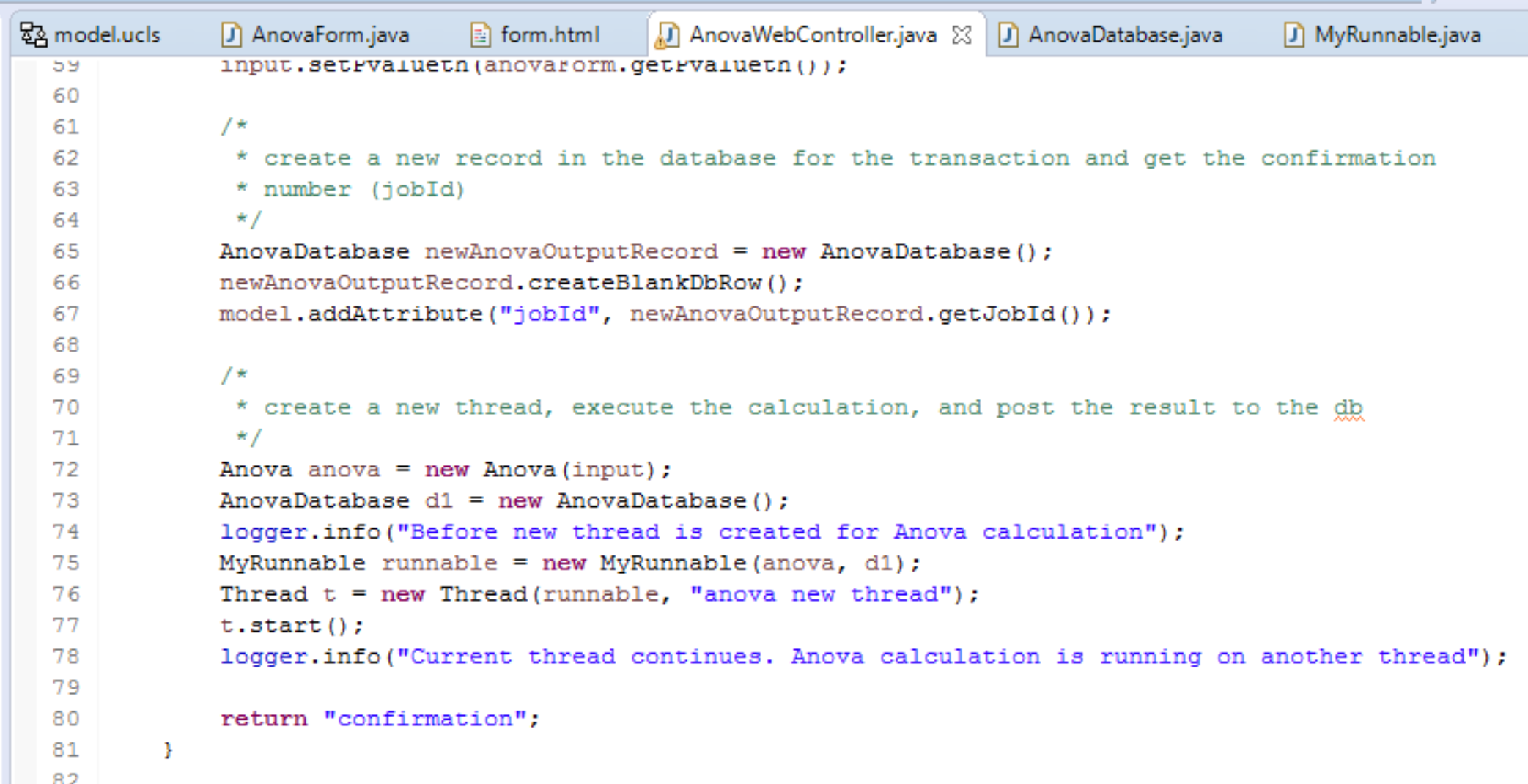


Figure 6: Creating a new thread and returning the confirmation page

Since the computation executes on a separate thread, the ‘confirmation’ page is returned. This page shows a simple message and job id.

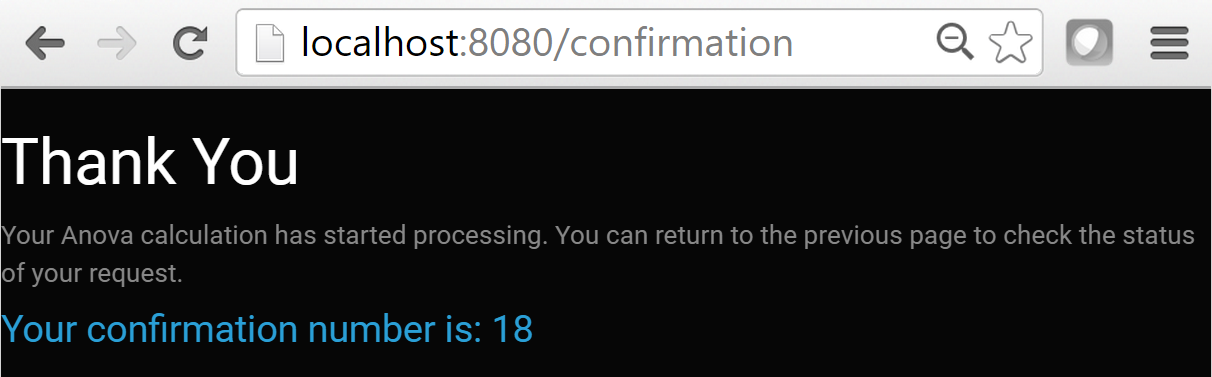


Figure 7: Confirmation page

### Query

The program also allows the user to find an existing ANOVA calculation by entering the job id (confirmation number).

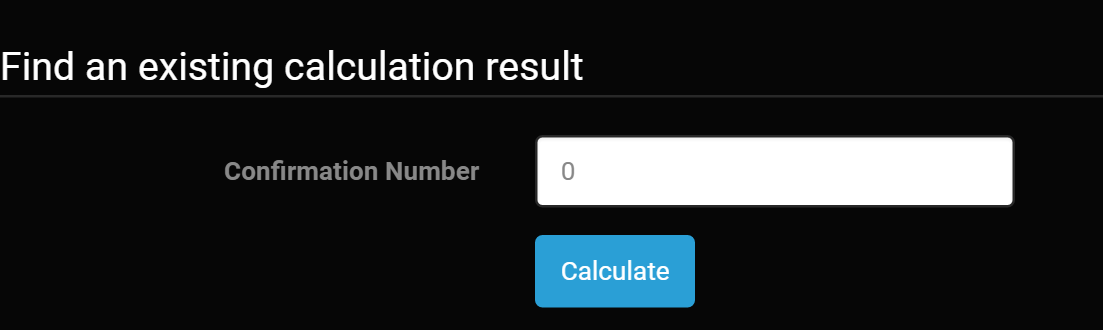


Figure 8: Query an existing calculation

To do this, I created another class to query the database for the values from the ANOVA output calculation.

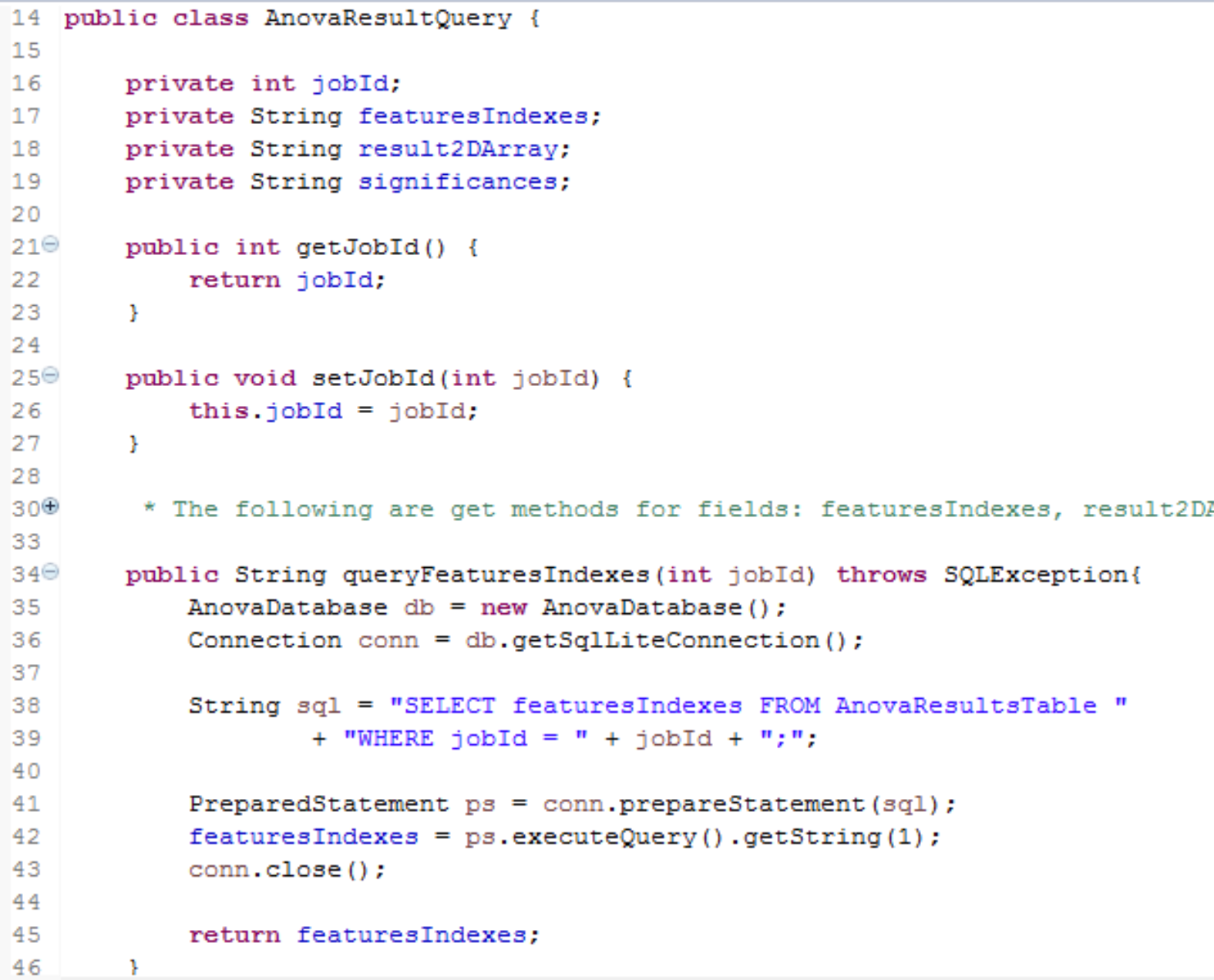


Figure 9: Querying the database for the output

Then upon HTTP GET request, given an input job id from the ‘form’, each field from the ANOVA output is added to the model. Once added to the model, I referenced them within the last View (‘query.html’).

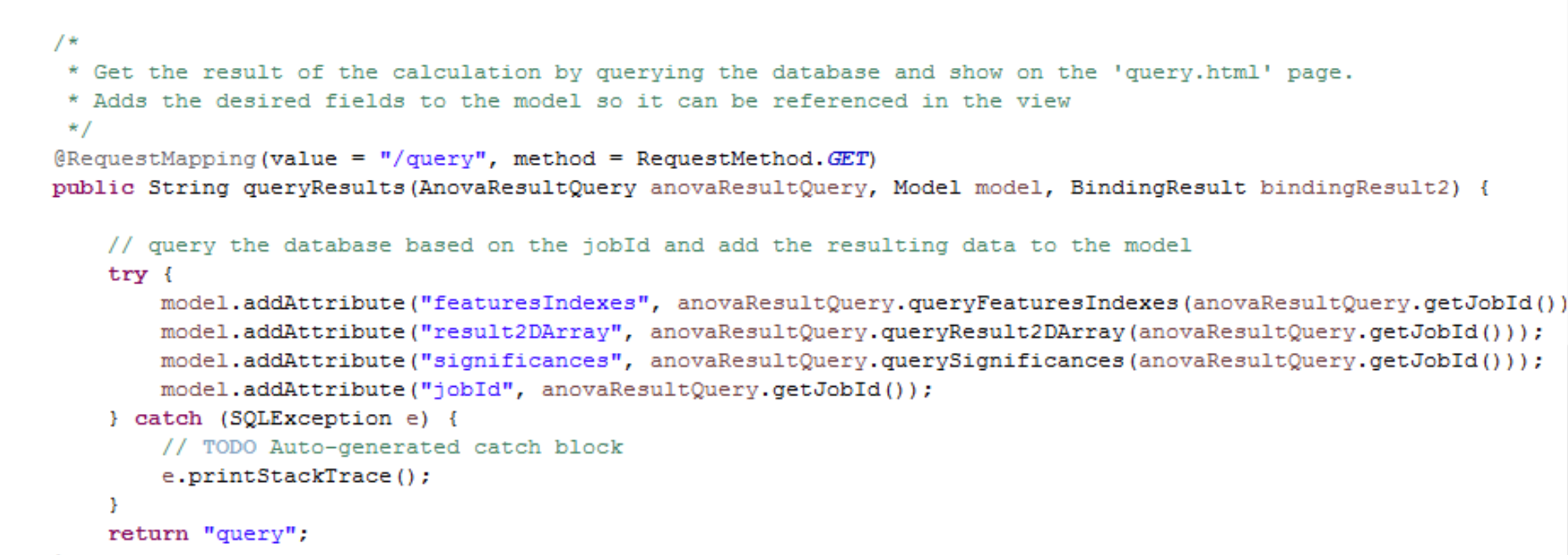


Figure 10: Adding the output to the model for View references

This program handles whether the computation is still pending or completed through the View.

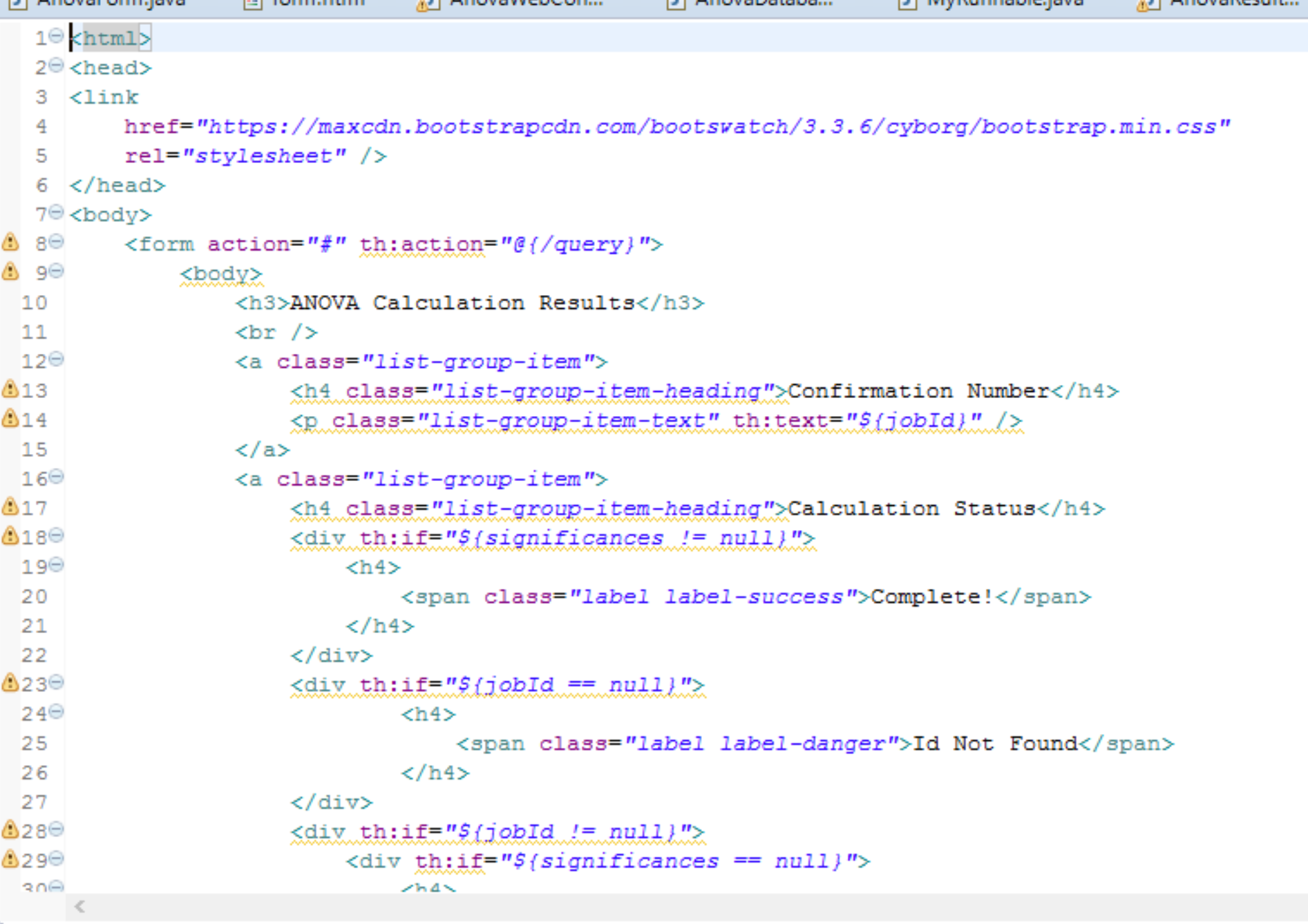


Figure 11: Representation of the output

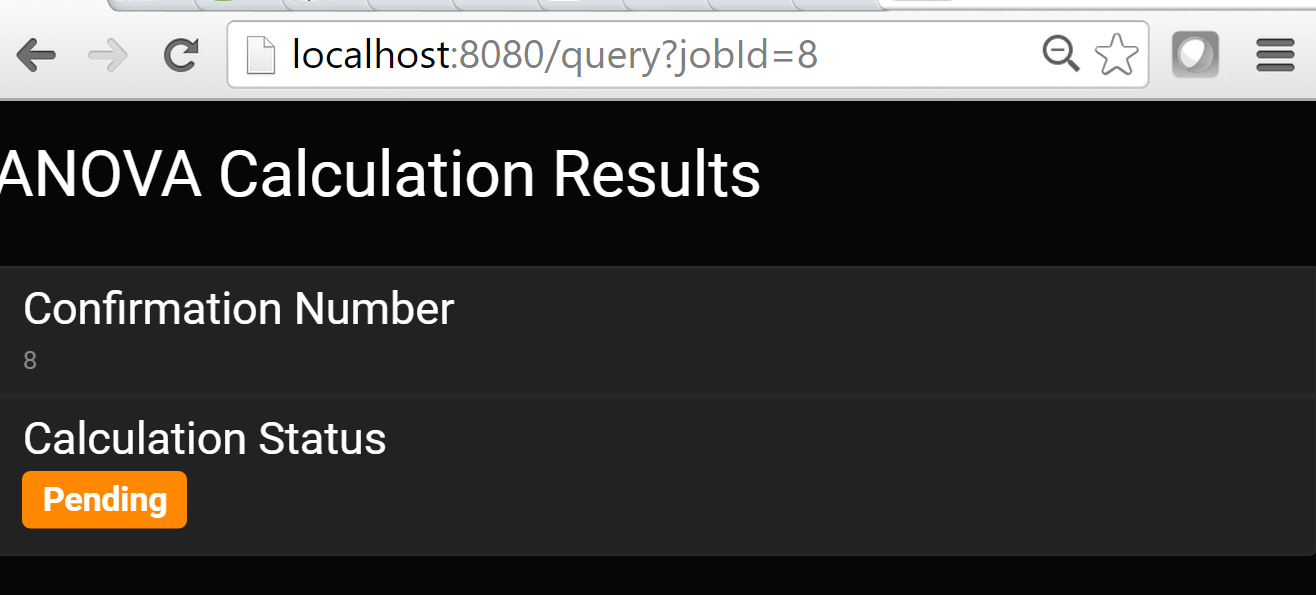
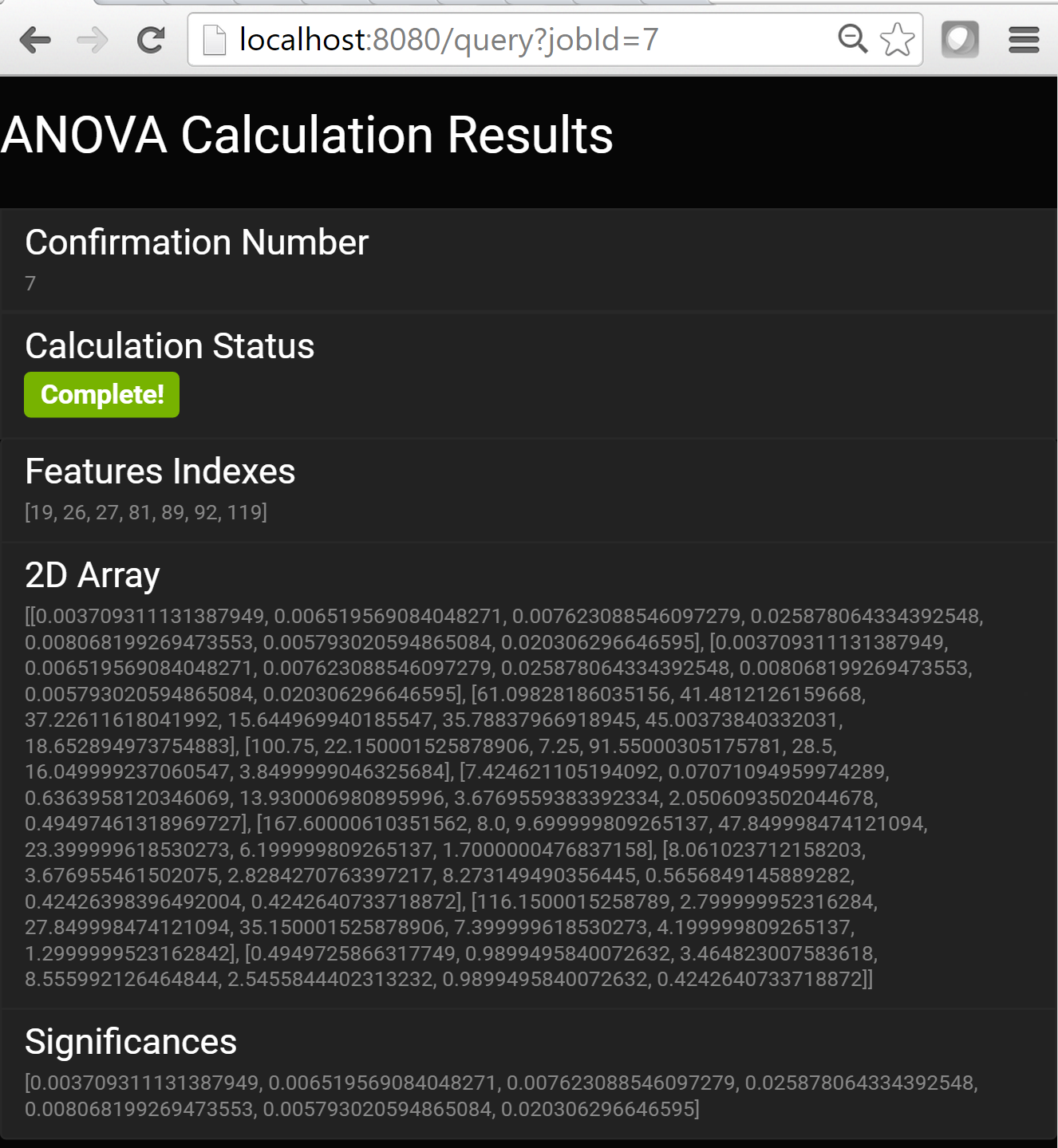


Figure 12: Sample outputs

# Conclusion

## Result

The Spring web service was able to perform an ANOVA calculation in a mostly RESTful fashion. It utilized the geWorkbench libraries and packages for execution, computed concurrently to prevent the program from being blocked using multiple threads, and provided functionality to query an existing calculation. Sample outputs from this program can be seen in the Appendix. The only regret from this project was that it did fully utilize JSON in rendering the final representation of the output. If this was done, the result of the calculation could have been better presented to the end user and also followed the ‘Representation’ REST principle. In addition, the input form could have been better for handling different scenarios of data entry. Finally, I believe I could have refactored this project and used less classes in a more efficient and clean manner. However, given time constraints, this was not possible and will need to be implemented for future versions of this program.

## Tools

* Language: Java
* Frameworks: Spring, Thymeleaf
* IDE: Eclipse
* Build: Maven
* ANOVA Calculation
  + Used packages from geworkbench project which also referenced external libraries
* Database: SQLite
* Code Repository: Git
* Version Control: GitHub app
* Reverse Engineered Class Diagram: http://www.objectaid.com/
* Debugging
  + Logging: org.slf4j.Logger
  + Testing: J-Unit
* CSS: Bootstrap ([*https://maxcdn.bootstrapcdn.com/bootswatch/3.3.6/cyborg/bootstrap.min.css*](https://maxcdn.bootstrapcdn.com/bootswatch/3.3.6/cyborg/bootstrap.min.css)*)*

# Appendix

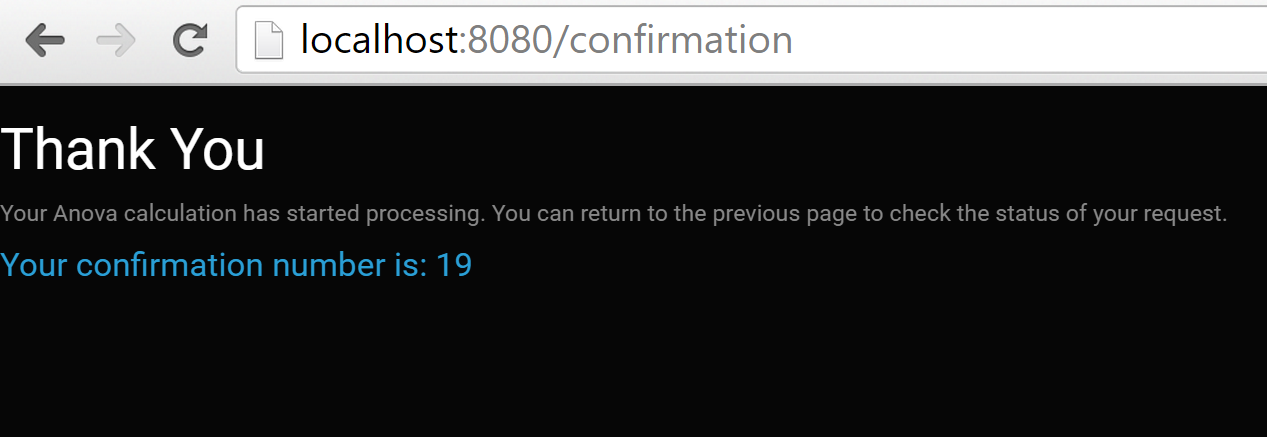
All code can be found on GitHub at: <https://github.com/rodgersk410/ANOVA-WebService>

## Web Service Sample

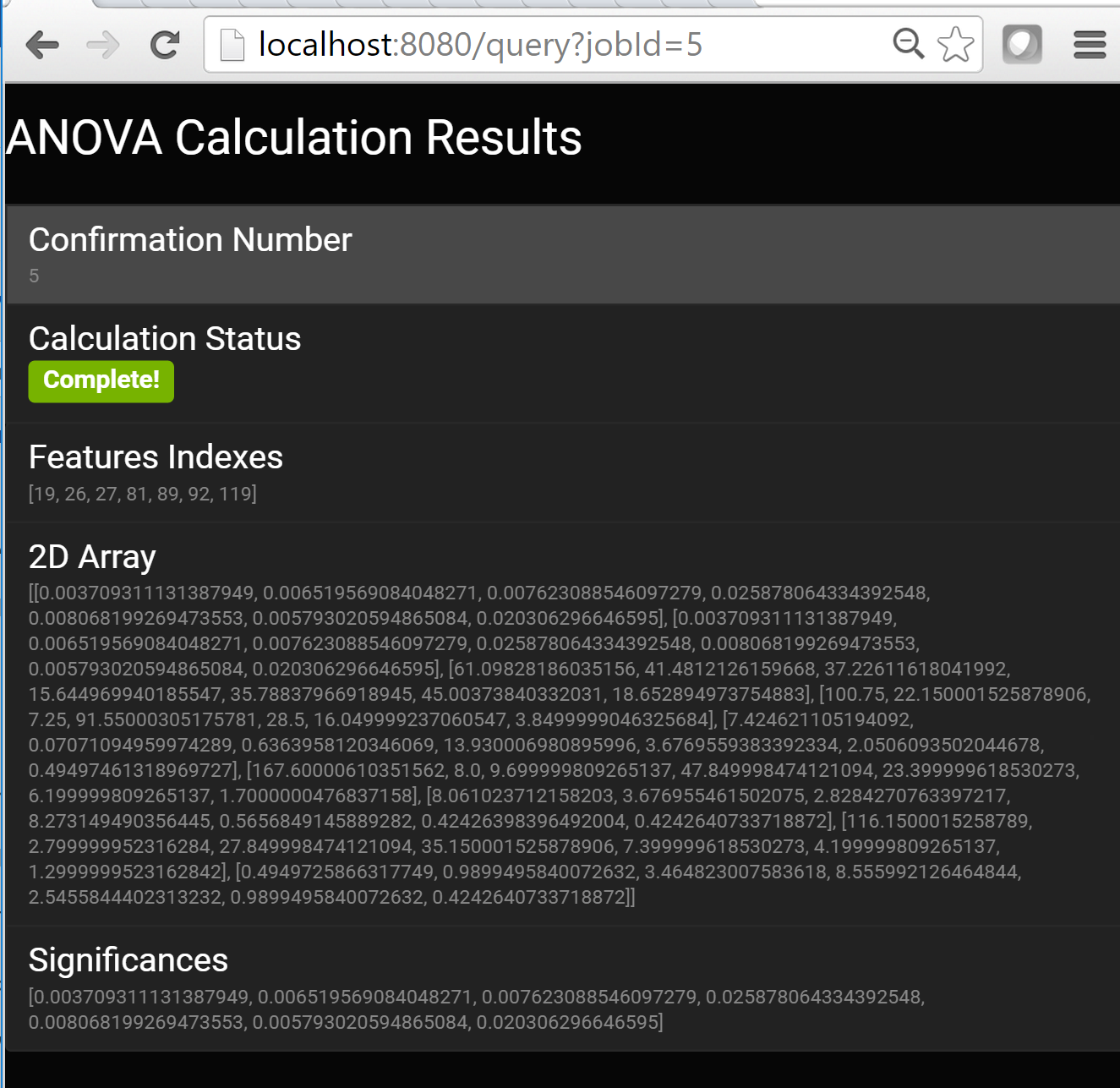
### Form



### Confirmation



### Query



# Class Diagram

The following is just a diagram of the classes I created for this program:

