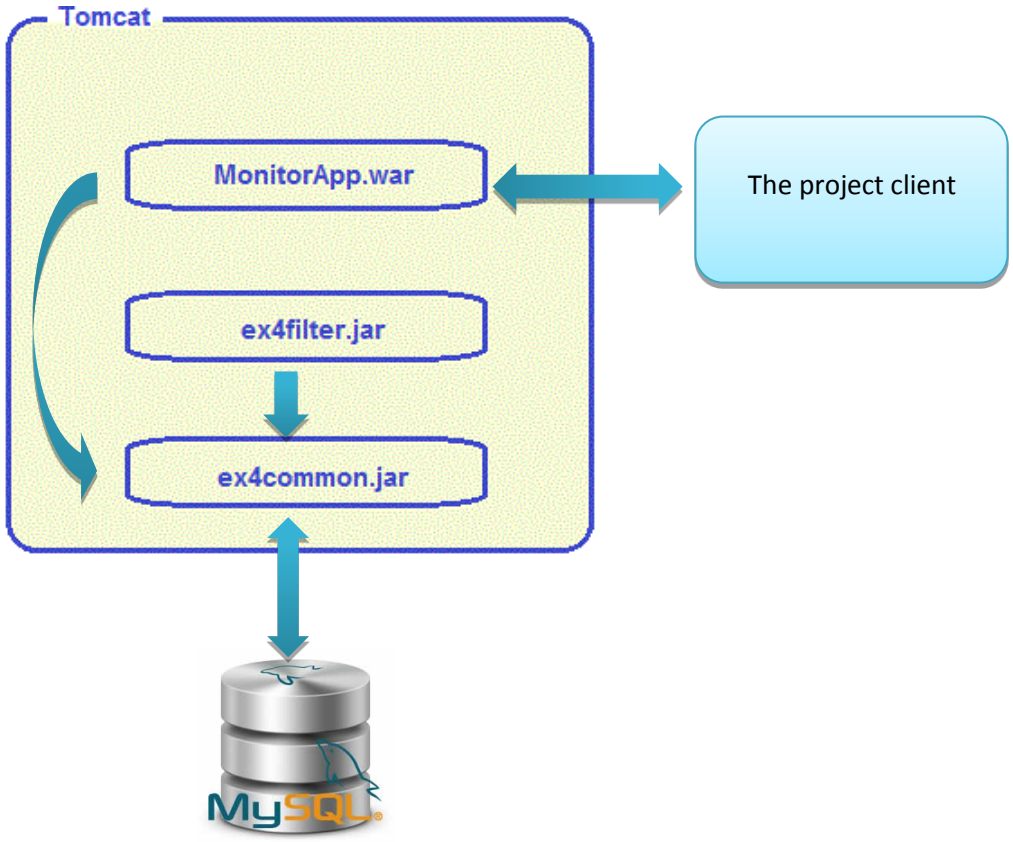
# Project structure in general

Our project contains four components:

1. ***ex4filter.jar*** – contains the filter.
2. ***MonitorWebApp.war***
3. ***ex4common.jar*** – manages the communication with the database.
4. **MySql DB**

The figure below describes the relationships between these components ( apologies for our poor drawing skills ☺ ) :




As you can see from the figure, the only component that "knows" the database is ex4common.jar. ex4common.jar is being used by ex4filter.jar for writing data into the database, and by MonitorWebApp.war for reading data from the database . When the filter comes across a data we wish to monitor, it uses the API of ex4common.jar for entering the data into the DB. When requests are made by the client to MonitorWebApp.war data need to be extracted from the DB, and it's being done by using the API of ex4common.jar as well.

# MySql DB design

*mta_db* scheme contains two tables:

1. **Applications** – contains names of the applications on Tomcat server. This table is created by executing:
```
CREATE TABLE IF NOT EXISTS mta_db.applications (id INT NOT NULL AUTO_INCREMENT PRIMARY KEY, name VARCHAR(64) ) ;
```
2. **PagesData** – contains all relevant data we need for a page resource. This table is created by executing:
```
CREATE TABLE IF NOT EXISTS mta_db.pagesData ( id INT NOT NULL AUTO_INCREMENT PRIMARY KEY, appName VARCHAR(64) NOT NULL, page VARCHAR(256) NOT NULL, method VARCHAR(10) NOT NULL, reqTime BIGINT NOT NULL, resDurationTime BIGINT NOT NULL, bytesSent BIGINT NOT NULL);
```

I used MySql Workbench tools to provide the following screen-shots:

Applications table:

| id | name |
|---|---|
| 1 | docs |
| 2 | examples |
| 3 | host-manager |
| 4 | manager |
| NULL | NULL |

This table contains all names of Tomcat applications that are being monitored by our filter.

PagesData table (edited to provide clear example):

| id | appName | page | method | reqTime | resDurationTime | bytesSent |
|---|---|---|---|---|---|---|
| 3 | /examples | /examples/jsp/jsp2/el/basic-comparisons.jsp | GET | 1319014909610 | 365 | 0 |
| 5 | /examples | /examples/jsp/jsp2/simpletag/hello.jsp | GET | 1319014912692 | 2250 | 0 |
| 6 | /examples | /examples/servlets/ | GET | 1319014921173 | 1 | 0 |
| 7 | /examples | /examples/servlets/servlet/RequestParamExample | GET | 1319014922680 | 2 | 665 |
| 8 | /examples | /examples/servlets/servlet/SessionExample | GET | 1319014925464 | 2 | 1138 |
| 9 | /docs | /docs/ | GET | 1319014931798 | 0 | 14651 |
| NULL | NULL | NULL | NULL | NULL | NULL | NULL |

This table contains the following data:

**Application name** – the name of the application that was accessed. (when a resources are retrieved for an application, the application name is being matched using regexp, as explained later on, so that doesn't matter that it starts with / in this table , but not in the application names table).
**Page (resource)** – the resources that were accessed by clients and matched our filter description (Post, Get, Servlets, JSPs).
**Method**– whether the resource was requested via a post/get request.
**reqTime** – the time of the request
**resDurationTime** – the time it took to process the request
**bytesSent** – the amoubt of bytes sent in the reponse. Relevant for servlets only. Will have the value 0 in JSPs.

# ex4common.jar

*ex4common.jar* contains a class called "*DBHelper*". *DBHelper* is solely responsible for managing the connection to the database.

This class holds the following private variable:

***private Connection conn;*** // A connection with a specific database

Creating an instance of this class causes the following functions to run:

1. ***public Connection initDBconnection()*** // initiates and returns the connection the mta_db scheme in MySql DB.

2. ***public void createApplicationsTableIfNotExist()*** // if not already exist, this function creates a table for the application names under mta_db scheme**.**
   We used the following SQL query for adding the Applications Table (we chose to use "IF NOT EXISTS" in our query as this method is being called as a part of the object instantiation process and it'll get instantiated once more in out WAR application) :

   ```
   CREATE TABLE IF NOT EXISTS mta_db.applications (id INT NOT NULL AUTO_INCREMENT
   PRIMARY KEY, name VARCHAR(64));
   ```

3. ***public void createPagesDataTableIfNotExist()*** // if not already exist, this function creates a table for the resources data under mta_db scheme.
   We used the following SQL query for adding the Pages Data table (we chose to use "IF NOT EXISTS" in our query as this method is being called as a part of the object instantiation process and it'll get instantiated once more in out WAR application) :

   ```
   CREATE TABLE IF NOT EXISTS mta_db.pagesData (id INT NOT NULL AUTO_INCREMENT
   PRIMARY KEY, appName VARCHAR(64) NOT NULL, page VARCHAR(256) NOT NULL, method
   VARCHAR(10) NOT NULL, reqTime BIGINT NOT NULL,resDurationTime BIGINT NOT NULL,
   bytesSent BIGINT NOT NULL);
   ```

This class contains several methods which can be divided to two groups:

1. methods that writes to DB:
   - `public void addRecordToApplicationsTable(String appName)` – when a new application is discovered, it's being added to the applications table by this method.
     The SQL query we use to add a record to the Applications tTable:

     ```
     INSERT INTO mta_db.applications (name) VALUES ("appName") ;
     ```

   - ```
     public void addRecordToPagesDataTable(String appName, String requestedResource,
                                 long requestTimeInMillis, long repsonseTimeInMillis,
                                 long bytesSent, String method) –
     ```
     When a request is made for a URI we wish to monitor, its data is being added to the pagesData table by this method.
     The SQL query we use to add a record to the Pages Data table:

     ```
     INSERT INTO `mta_db`.`pagesdata` ( `appName`, `page`, `method`, `reqTime`,
     `resDurationTime`, `bytesSent` ) VALUES
     ( 'appName', 'requestedResource', 'method', 'requestTimeInMillis',
     'repsonseTimeInMillis', 'bytesSent' ) ;
     ```
     ```

2. methods that reads from DB:
    - ***public List <String> getApplications()*** – returns a list with all the application names in the application table.
      We use the following SQL query for retrieving the list:

      ```
      SELECT DISTINCT name FROM mta_db.applications;
      ```

    - ***public List <String> getResouecesForApplication()*** – returns all the resources (URIs) for a given application name. The SQL query we use to get the resources of an application:

      ```
      SELECT DISTINCT page FROM mta_db.pagesdata where appName REGEXP '^/?appName/?$';
      ```

      We used REGEXP in the query in order to match application names that start/end with/without "/".

    - ***public double [] getDataForResourceByMethod (String resource , String method)*** – for a given URI and method (get/post) , it returns an array of size 3 which contains: average response time, requests per minute, average bytes per minute (the last cell is ignored when the URI is a jsp).

      ```
      SELECT avg(resDurationTime), count(page), avg(bytesSent) from mta_db.pagesData
      where method = 'method' and ( reqTime between 'minuteAgoTime' and 'currTime' )
      and page regexp '^/?resource/?$' ;
      ```

      We used REGEXP in the query in order to match resources that start/end with/without "/".

# ex4filter.jar

Class *MonitorFilter* is responsible for filtering the http requests that are sent to Tomcat server.

Resources that are relevant to our project are servlets and JSPs that are requested via post / get requests.

For each relevant resource we collect the following data:

1. The name of application that is destined to receive the request.
2. The path of the resource on the application that is destined to receive the request (URI).
3. The request time (using *currentTimeMillis*).
4. The time it took to serve the request (in milliseconds).
5. Amount of bytes sent as reply (for servlets only).

Upon deciding a certain request is relevant to us, *MonitorFilter* uses an instance of class *DBHelper* in order to insert the data to the DB by calling its method *addRecordToPagesDataTable*.

# MonitorWebApp.war

"MonitorWebApp.war" composed by the following elements:

- **DBWrapper** – the DBWrapper is a singleton class for the DBHelper. We figured that we don't want to overload "Tomcat" web server with many MySQL connections. To resolve this problem we wrapped DBHelper class with a singleton class which creates a single DBHelper instance and publishes only 3 methods of the DBHelper which are essential to the "monitoring servlet", which are:

  1. ***List <String> getApps()–*** This function return a list of application running on the web server.

  2. ***List <String> getResources (String appName)*** – Return a list of resources of a specific application ('appName').

  3. ***getDataForResourceByMethod(String Resource, String Method)*** – Returns an array with information on the resource ('Resource'), like average request per minute, etc…

  *By doing so, our war application uses only one connection to the database*. Since a monitor application isn't usually meant to be widely used by many users it seemed like the logical thing to do (at least in our case).

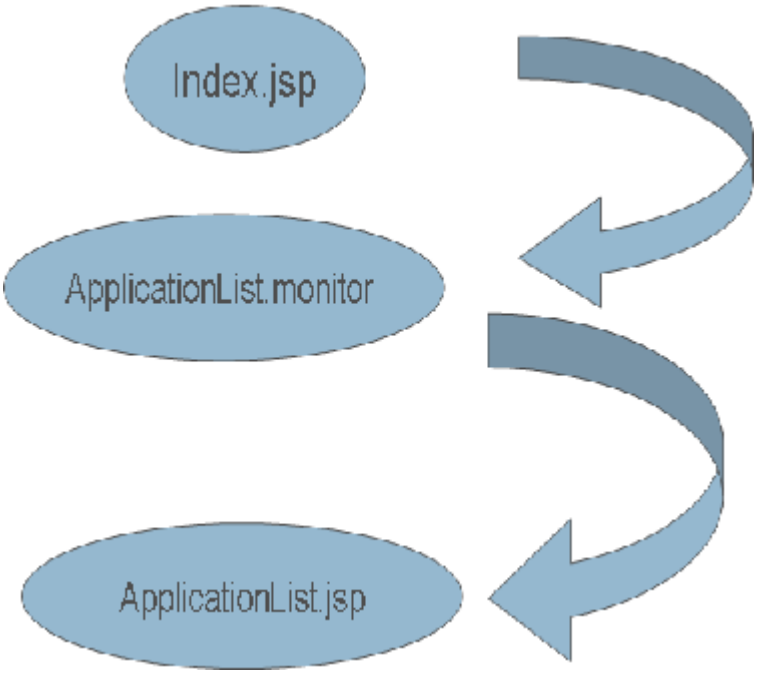
- **MonitorWebApp** – An HTTP servlet running on the web server. This servlet wraps any HTTP request (GET / POST) for "*.monitor" resource. GET and POST request are handled the same. When some request arrives to the servlet, the servlet first extracts the requested resource and handle the request only if the resource is supported by the servlet.

  The servlet supports 3 "*.monitor" resource:

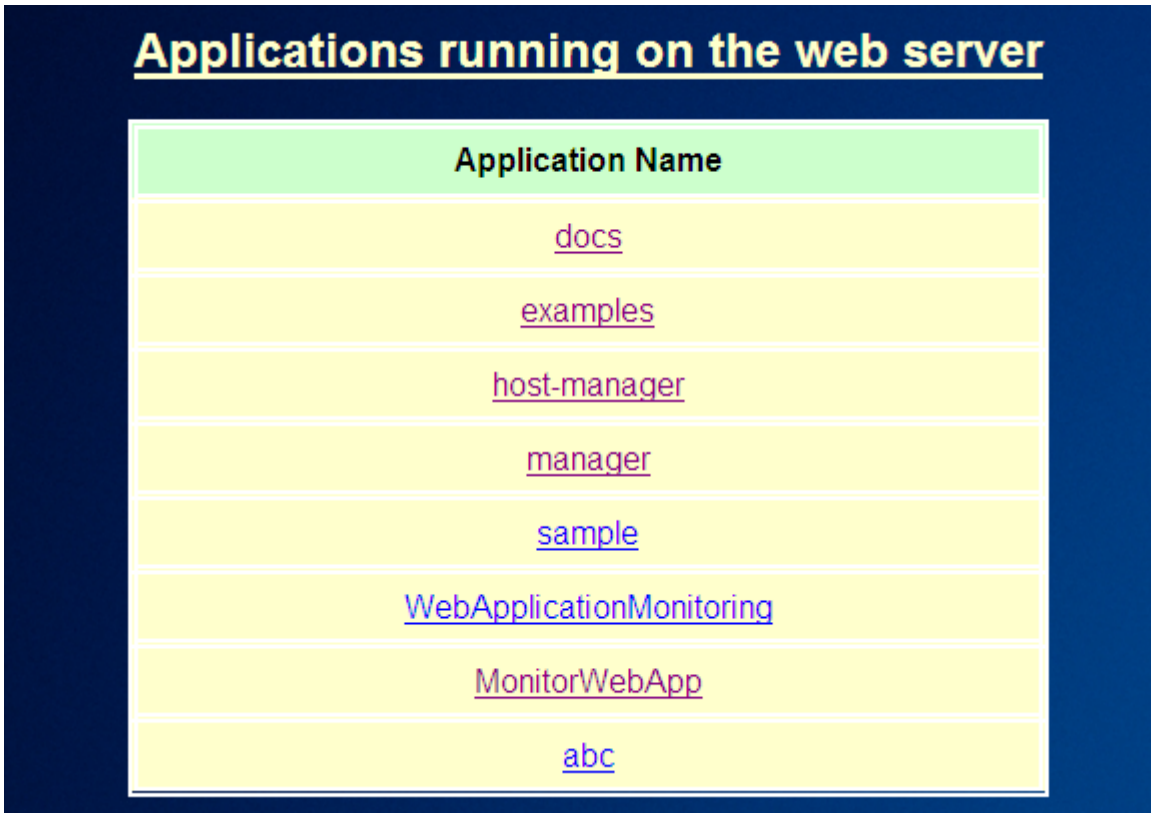
  1. **ApplicationList.monitor**:

     First, "index.jsp" makes a redirection to this resource. The servlet then catches this request and creates an application list. Afterward, it forwards the request to the "ApplicationList.jsp" where the application list is presented in a HTML table.

     - The following figure shows the flow of the events.

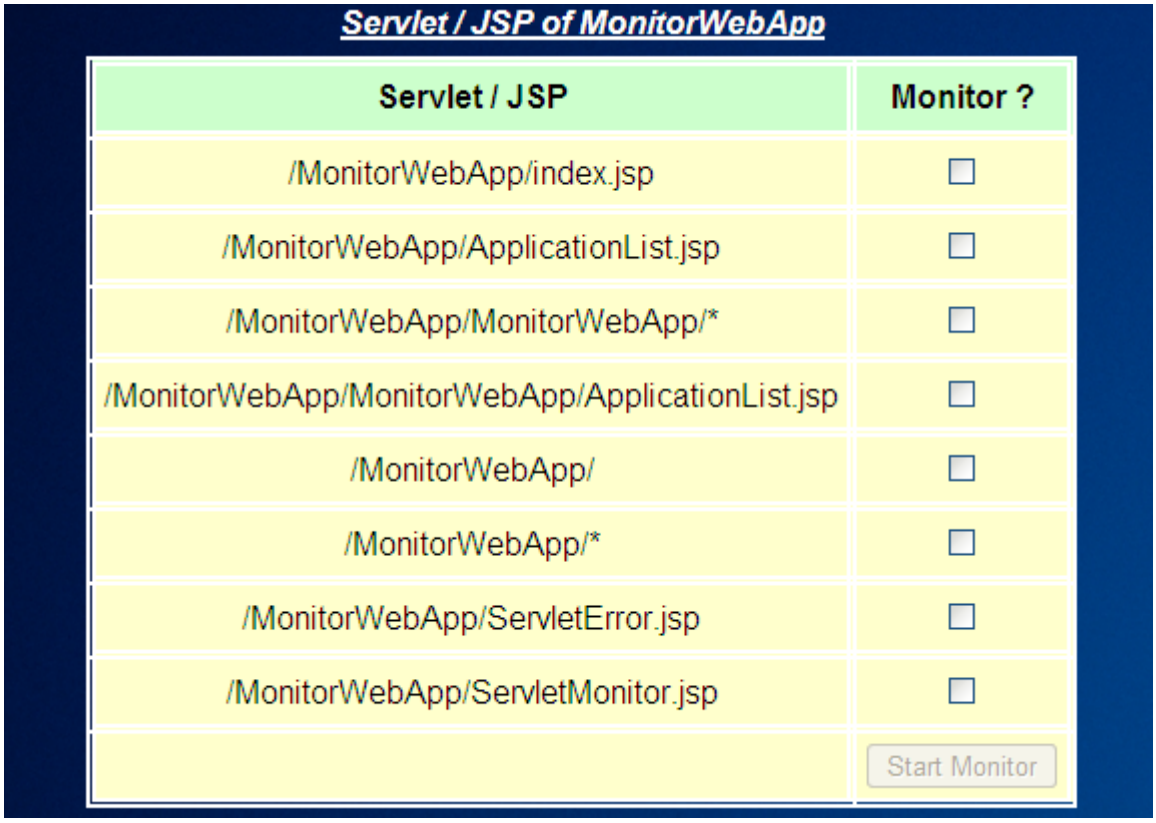


- The following figures shows an example of “ApplicationList.jsp” output




2. **ServletList.monitor**:

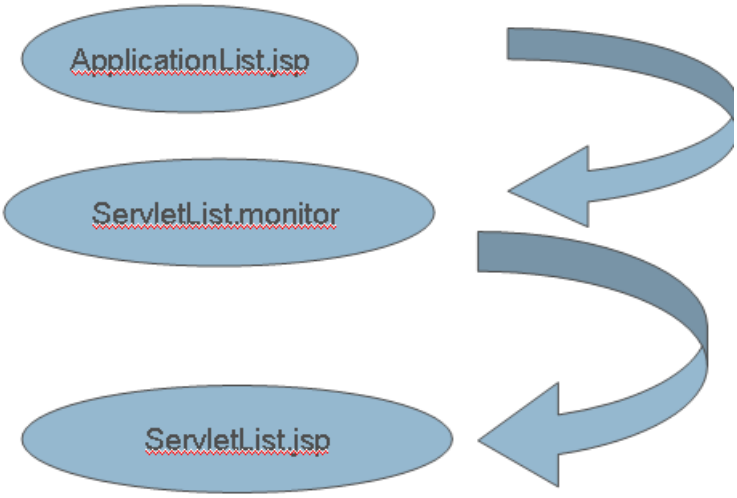
   When selecting an application from “ApplicationList.jsp” page, the browser makes an HTTP request, where the application name is sent using a QueryString. The servelt catches the request, extracts the application name from the request’s QueryString, and calls the *getResources()* function and send the resource list of the selected application to “ServletList.jsp”, where it shows in a table. See example below…



As you can see, the user gets a list of resource of his / her desired application and he / she can start monitoring some / all the resources by selecting the relevant checkbox.

Please note that the monitor button is disabled by default. The monitor button will be enabled only when at least only resource is selected (it being verified by js code).

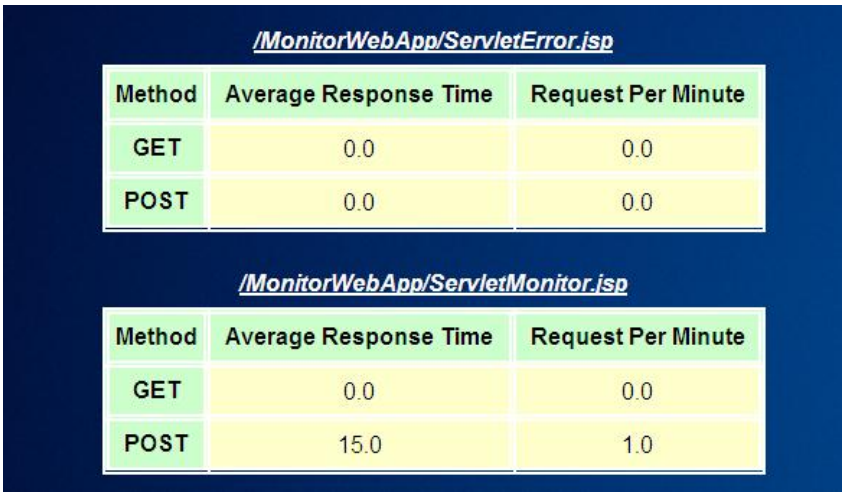
The following figure shows the flow from the “ApplicationList” page to the “ServletList” Page.




3. **ServletMonitor.monitor**:

   When the user selects some resource to monitor and clicks the “Start Monitor” button, the browser makes a POST request, posting each and every resource he / she wants to monitor to “ServletMonotor.jsp”. This page then dynamically builds a table for each resource. Each table shows the following statistics for each chosen resource:

   - Average Response Time (ms)
   - Request Per Minute
   - Bytes Per Minute (only for servlets, not jsp files)



*/MonitorWebApp/ServletError.jsp*

| Method | Average Response Time | Request Per Minute |
|---|---|---|
| GET | 0.0 | 0.0 |
| POST | 0.0 | 0.0 |

*/MonitorWebApp/ServletMonitor.jsp*

| Method | Average Response Time | Request Per Minute |
|---|---|---|
| GET | 0.0 | 0.0 |
| POST | 15.0 | 1.0 |

   The resource statistic tables are updated every 10 seconds using AJAX . Every 10 seconds the page makes several asynchronic requests (single request for each resource). The requests are made to “ServletMonitor.monitor”. Again, our servlet catches this request, extracts the request resources, get the statistic for the relevant resource (for POST & GET methods), assembles the statistics to an XML format and send the XML output back to the AJAX object. The Ajax callback function receives the XML info, parses it and presents the information back to the relevant table.

- The XML resource information format is as following:

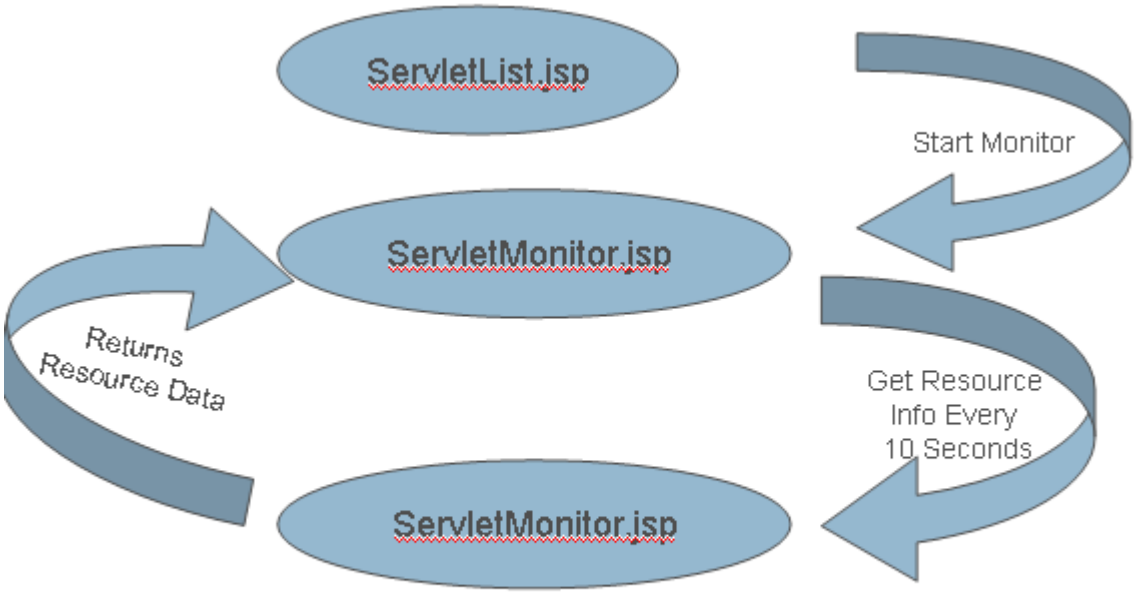
```
</ResourceInfo>

   <Resource>
      <Name> Resource Name </Name>
      <Method> GET </Method> +
      <AverageResponseTime> DATA"</AverageResponseTime>
      <RequestPerMinute> DATA </RequestPerMinute>\n" +
      <AverageBytesPerMinute> DATA </AverageBytesPerMinute>
   </Resource>

   <Resource>
      <Name> Resource Name </Name>
      <Method> POST </Method> +
      <AverageResponseTime> DATA"</AverageResponseTime>
      <RequestPerMinute> DATA </RequestPerMinute>\n" +
      <AverageBytesPerMinute> DATA </AverageBytesPerMinute>
   </Resource>

</ResourceInfo>
```

- The following figure shows the flow from the moment where the user selects a resources to monitor.



***The content of the application's web.xml file:***

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns="http://java.sun.com/xml/ns/javaee"
	xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
	id="WebApp_ID" version="3.0" metadata-complete="true">
	<display-name>MonitorWebApp</display-name>
	<welcome-file-list>
		<welcome-file>index.jsp</welcome-file>
	</welcome-file-list>

	<servlet>
		<servlet-name>MonitorWebApp</servlet-name>
		<servlet-class>model.MonitorWebApp</servlet-class>
	</servlet>

	<servlet-mapping>
		<servlet-name>MonitorWebApp</servlet-name>
		<url-pattern>*.monitor</url-pattern>
	</servlet-mapping>

</web-app>
```

As you can see, we set index.jsp to be the welcome file of our application, meaning – when accessing the application via "localhos:8080/MonitorWebApp/" the user will get to "localhos:8080/MonitorWebApp/index.jsp"

We also set our servlet to monitor URLs the fit the pattern *.monitor , as we explained earlier.

# Real monitoring product – things to consider

Our project, of course, was not meant to give a full solution for a monitoring product.

In real life, a monitoring product needs to deal with a great load of data and heavy traffic to the monitored applications.

From a DB management perspective, the more data a DB contains – the more time it takes to execute a query. In real life monitoring environment, we would have to decide criteria to how much data we store in the DB. For example, suppose we know we only need to be able to monitor the last year – we would like to delete the irrelevant data from the DB once in a while to speed it up (not really delete, probably archive it to allow further use of this data if needed in the future.) In our project we needed only data of the last 60 seconds, but since we handled very small amount of data, it was unnecessary to take care of deleting/archiving the irrelevant data.

From a DB design perspective, in a real full product, as we said, the DB needs to hold a great amount of data. It's possible to plan the DB tables in a more efficient way that will speed up query executions. (For example, by not repeating the application name in two tables like we did, but use a key to connect between the two). However, due to the expected small amount of data in our DB, such modifications will achieve minor benefit if any, so we went for more convenient design.

Another thing we struggled with was the number of DB connection we wish to allow. Of course, MySql is designed to handle many connections at once. But since our project is done for learning purposes and we had to create the DB tables ourselves in our code (as opposed, for example, to using a script once to create them by the IT), each instantiation of out DBHelper project, also executed an SQL query to create the tables if they don't exist. We wanted to minimize these queries, so we used only one instance of DBHelper it in our MonitorWebApp.war application. We assumed only one user uses this application anyway so we preferred saving some queries. In a real product, there may be more than one user of the monitoring application, and the table creation will not be executed from the code as we did, so we believe it would be best to create a DB connection for each user, as the synchronization issue is being taken care of by MySql.