**Load Testing with SoapUI**

**Pre requisite**

* Spring Boot
* Soap-UI
* MySQL

**LoadTesting**

Load testing is a [non-functional testing](http://www.softwaretestingclass.com/functional-testing-vs-non-functional-testing/) of a software application where the system is loaded with number of requests in order to measure the system responses.

Load testing is performed for an application with the anticipated volume in production to make sure that there is no deviation in the system response or its behavior while operating with this volume and such application can be marked as safe.

Load testing is also known as volume testing and software performance testing.

**Sample Simple Spring Boot Application with Data base Connectivity**

1. **Sample**

**Project Structure**

├──src

│   └──main

│   ├──java

│   │    └──com

│   │    └──tech

│   │   └──cadt

│ │    ├──ApplicationConfig.java

│ │    ├──OrderRepository.java

│ │    ├──OrderService.java

│ │ │

│ │ └──order

│ │ ├──entity

│   │ │ └──OrderProduct.java

│ │ │

│ │ └──rest

│ │ └──OrderRestServiceController.java│ │

│ └──resources

│ └──application.properties

│

│

└── pom.xml

**Pom.xml**

|  |
| --- |
| <project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>  <modelVersion>4.0.0</modelVersion>  <groupId>Sample</groupId>  <artifactId>Sample</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>jar</packaging>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.4.1.RELEASE</version>  </parent>    <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  </properties>    <dependencies>    <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <scope>runtime</scope>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  <version>3.1.0</version>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>    <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jersey</artifactId>  </dependency>    <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>    <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>    <dependency>  <groupId>com.h2database</groupId>  <artifactId>h2</artifactId>  <scope>runtime</scope>  </dependency>    <dependency>  <groupId>org.json</groupId>  <artifactId>json</artifactId>  <version>20090211</version>  </dependency>  </dependencies>  </project> |

|  |
| --- |
| **ApplicationConfig.java** |
| **Path 🡪 src/main/java/com/tech/cadt/ ApplicationConfig.java** |
| **package** com.tech.cadt;  **import** javax.inject.Named;  **import** org.glassfish.jersey.server.ResourceConfig;  **import** org.springframework.context.annotation.Configuration;  @Configuration  **public** **class** ApplicationConfig {  @Named  **static** **class** JerseyConfig **extends** ResourceConfig {  **public** JerseyConfig() {  **this**.packages("com.tech.cadt.\*");  }  }  } |

|  |
| --- |
| **OrderRepository.java** |
| **Path 🡪 src/main/java/com/tech/cadt/ OrderRepository.java** |
| package com.tech.cadt;  import java.util.List;  import org.springframework.data.repository.CrudRepository;  import org.springframework.transaction.annotation.Transactional;  import com.tech.cadt.order.entity.OrderProduct;  @Transactional  public interface OrderRepository extends CrudRepository<OrderProduct, Integer> {  public List<OrderProduct> findByOrderId(Integer orderId);  } |
| **Path 🡪 src/main/java/com/tech/cadt/ OrderService.java** |
| **OrderService.java** |
| package com.tech.cadt;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.context.annotation.ComponentScan;  import org.springframework.stereotype.Component;  @SpringBootApplication  @EnableAutoConfiguration  @Component  @ComponentScan("com.tech.cadt.order.rest")  public class OrderService {  public static void main(String[] args) {  SpringApplication.run(OrderService.class, args);  }  } |

|  |
| --- |
| **OrderProduct.java** |
| **Path 🡪 src/main/java/com/tech/cadt/ order/entity/OrderProduct.java** |
| package com.tech.cadt.order.entity;  import java.io.Serializable;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.Table;  @Entity  @Table  public class OrderProduct implements Serializable{  @Id  @GeneratedValue(strategy = GenerationType.AUTO)    private Integer orderId;    private String username;    private String paymentMode;    public Integer getOrderId() {  return orderId;  }  public void setOrderId(Integer orderId) {  this.orderId = orderId;  }  public String getUsername() {  return username;  }  public void setUsername(String username) {  this.username = username;  }  public String getPaymentMode() {  return paymentMode;  }  public void setPaymentMode(String paymentMode) {  this.paymentMode = paymentMode;  }  } |

|  |
| --- |
| **OrderRestServiceController.java** |
| **Path 🡪 src/main/java/com/tech/cadt/ order/rest/ OrderRestServiceController.java** |
| package com.tech.cadt.order.rest;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.http.HttpHeaders;  import org.springframework.stereotype.Component;  import org.springframework.web.bind.annotation.CrossOrigin;  import org.springframework.web.bind.annotation.RequestBody;  import org.springframework.web.bind.annotation.RequestHeader;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.RestController;  import com.tech.cadt.OrderRepository;  import com.tech.cadt.order.entity.OrderProduct;  @RestController  @RequestMapping("/orders")  @CrossOrigin  @Component  public class OrderRestServiceController {  @Autowired  OrderRepository orderRepository;  @RequestMapping(value="/create", method = RequestMethod.POST)  @CrossOrigin  public OrderProduct createOrder(@RequestBody OrderProduct order,HttpServletResponse httpServletResponse,@RequestHeader HttpHeaders headers,HttpServletRequest httpServletRequest) {  System.out.print("inside create");  order.getOrderId();  order = orderRepository.save(order);  return order;  }        } |

**Database connectivity in application.properties**

|  |
| --- |
| **application.properties** |
| **Path 🡪 src/main/resources/application.properties** |
| server.port = 8887  server.host = localhost  spring.datasource.url=jdbc:mysql://localhost:3306/samples  spring.datasource.username=root  spring.datasource.password=root  spring.jpa.hibernate.ddl-auto=update |

* Start mysql server
* Create database with the name samples

**LoadTesting using SOAP-UI**

1. Click on REST in the Soap-UI and enter the webservice URL

For example : <http://localhost:8887/orders/create>

Change method to Post and send the input values in json format

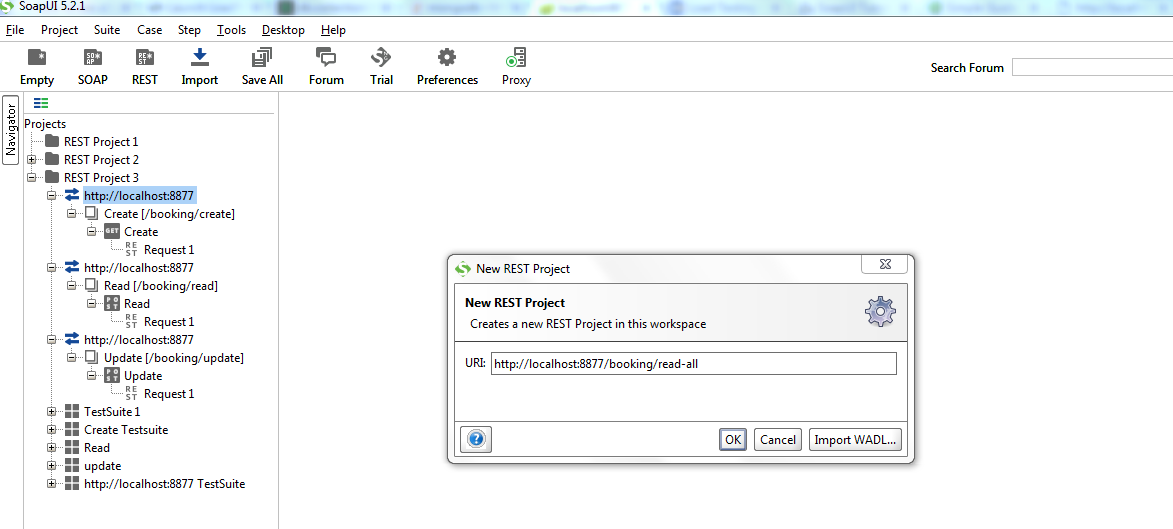
Eg:

{

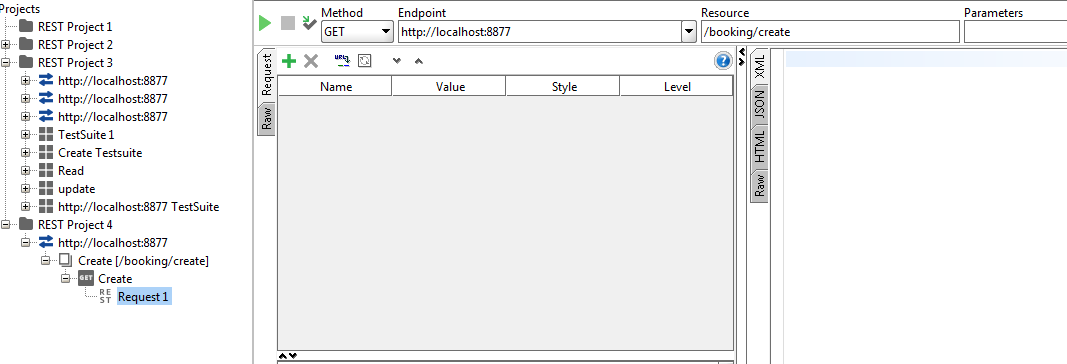
"username":"cadt",

"paymentMode":"self"

}

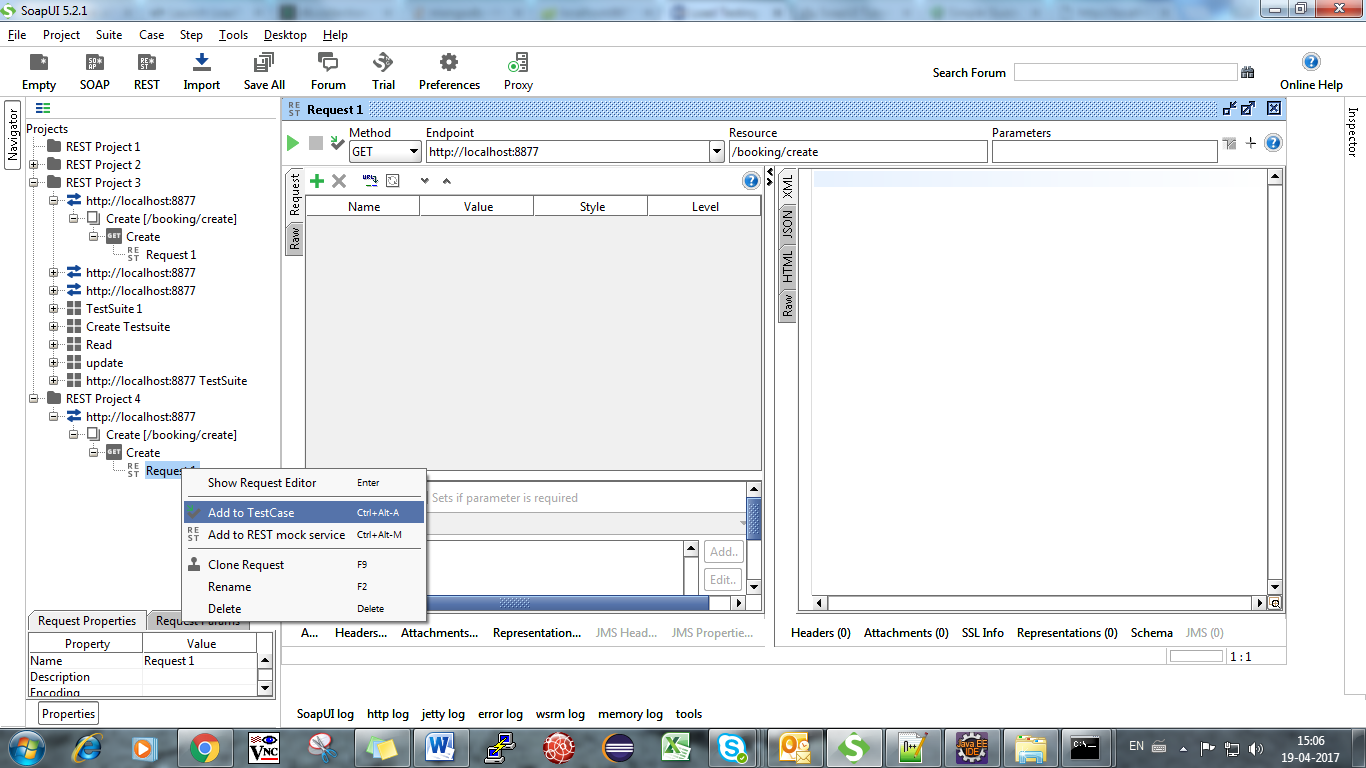


It will create new REST project

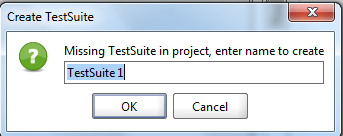


1. Creating Testing Suite

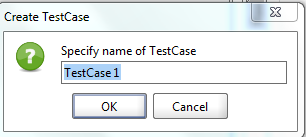
Right click on the request ->Add to Testcase



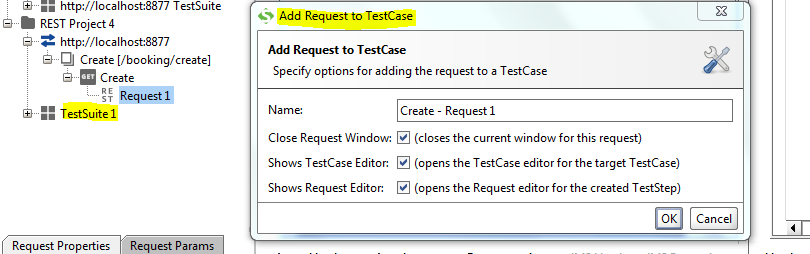
Create TestSuite->add name for the test suite and click OK



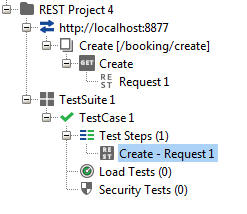
Create TestCase->Specify name of test case and click OK



Test Suite will be created ,Add request to testcase -> click OK

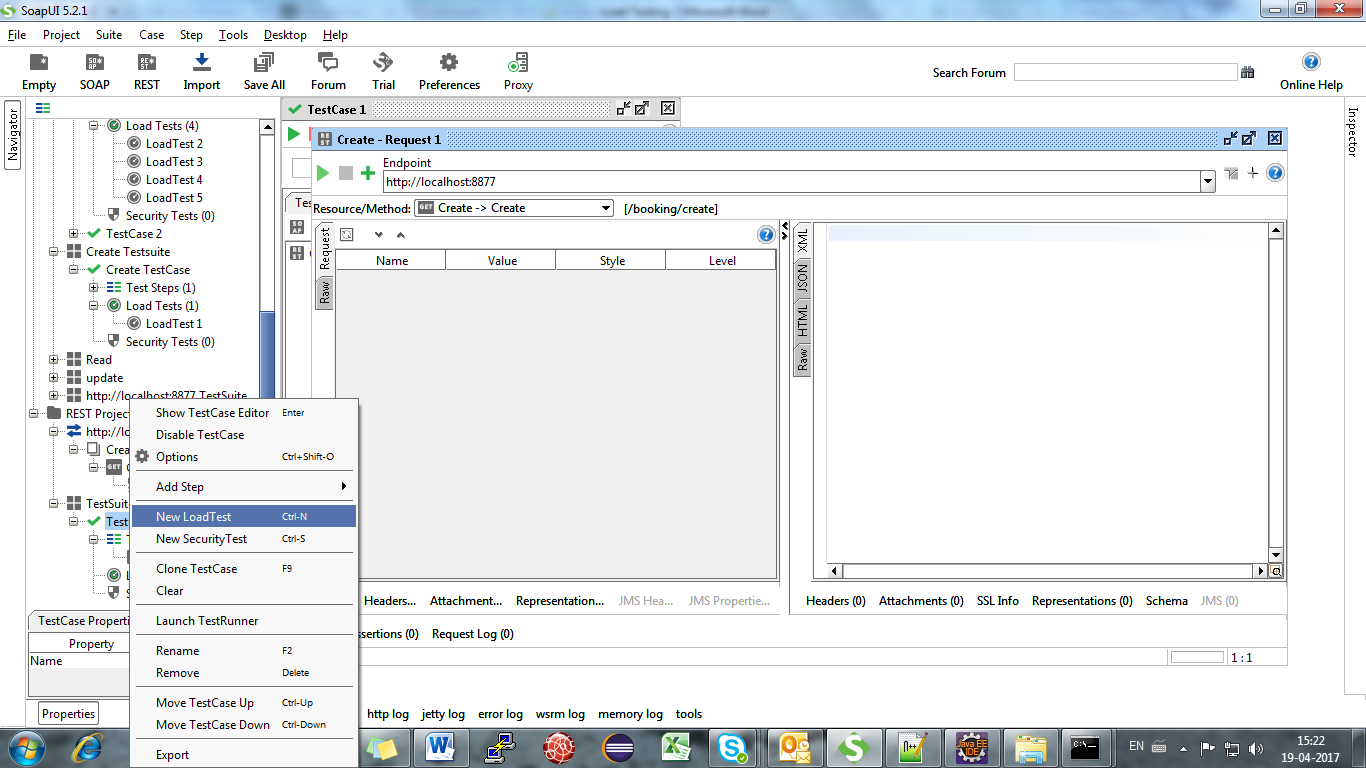


Test Suite Created

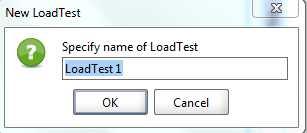


1. Creating Load Test

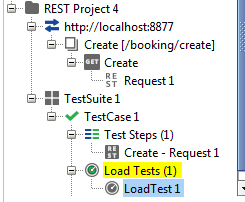
Right Click on TestCase under TestSuite and click-> New Load Test



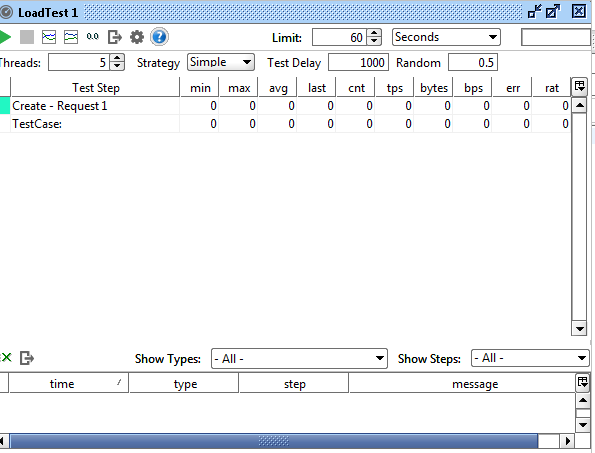
Mention the Load test Name and click ->OK



Now the Load Test has been created



Now click on Load Test1



Load test window will open

The first icon in the top left corner is a “Play” button icon that runs the load test. The second icon is used to stop the current execution. The next 4 icons are related to Statistics graphs, and then there is the option icon. Following those icons is the icon used to convert this test into a loadUI test case, and the last icon is help.

Limit will allow you to set the limit in seconds, total runs, or runs per thread, so you can choose how your test will run.

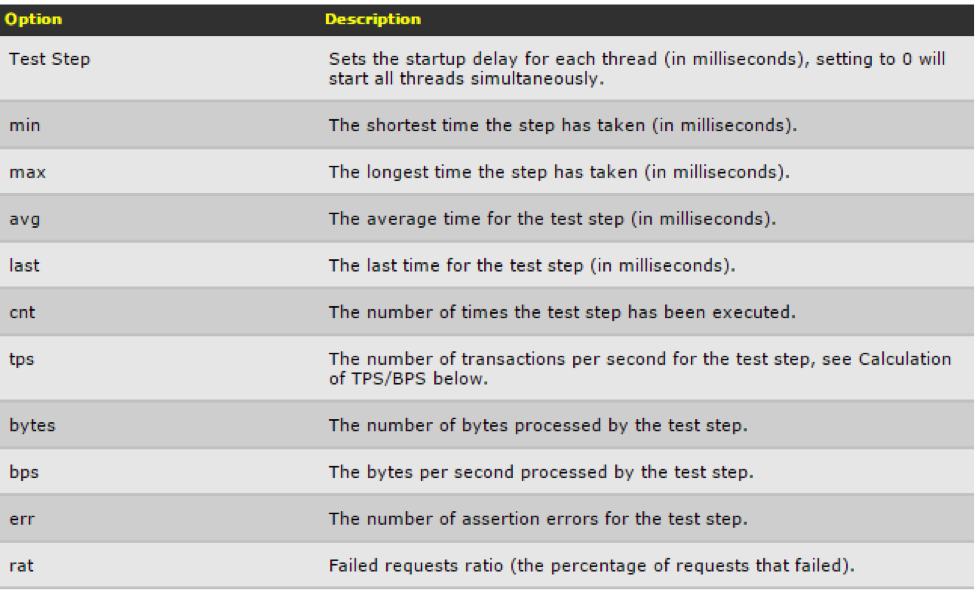
The next Box shows the percentage of the current test execution.

Threads allow us to set the number of virtual users we want to run this test. The default value is 5.

Strategy – This is how we want to run the load test and it should be chosen based on the requirements. Let’s see the different strategies and their significance.

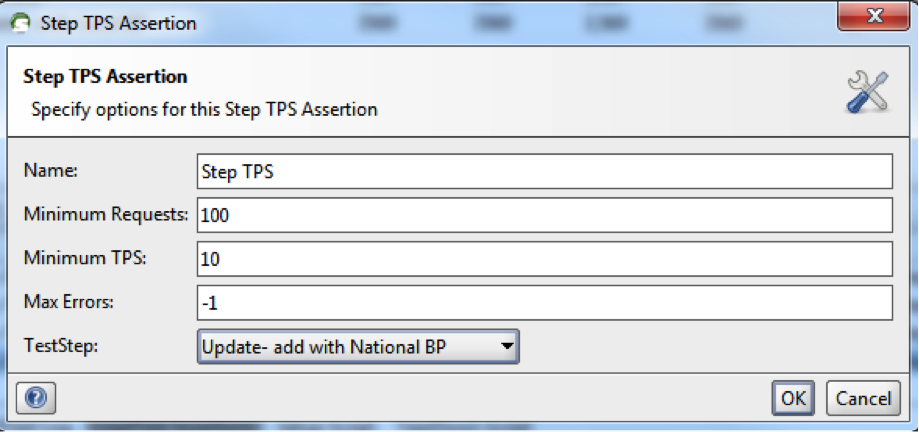
* Simple – This is the default strategy to be used. It runs the test case with the number of visual users defined in the “Threads” and you can also define the delay in between each run.  The “Test Delay” field lets us define the delay in milliseconds. “Random” defines the order of randomness you want in the delay, with “0” meaning no randomness and “1” meaning all random. This strategy is good for running the baseline tests.
* Burst – In this strategy, it runs all the defined threads at once for the time specified into “Burst Duration” and then it waits for the time defined into “Burst Delay”. After that, it will burst all threads again. It’s good to test the recoverability of the application.
* Variance – In this mode, the test runs with the specified threads for the duration that was specified into “Interval” in seconds and the thread count varies ups and downs (like ECG readings) with the order defined into the “Variance”. So if we set the number of threads to 6, the Interval to 120 seconds, and the variance to 0.5, then the test will run for 120 seconds with threads ranging from 3 to 9. This strategy is good for stress testing, as well as running to see the baseline test once we have done the “Simple” Strategy for baseline.
* Thread – This strategy is used to observe the AUT behavior. The virtual users increases by an increment of 1, starting from the value set at “Start Threads” and continuing until “End Threads” within the time specified into Limit. It usually used to find the breaking point.

When the test is running, the statistics are collected and displayed under the following headers:

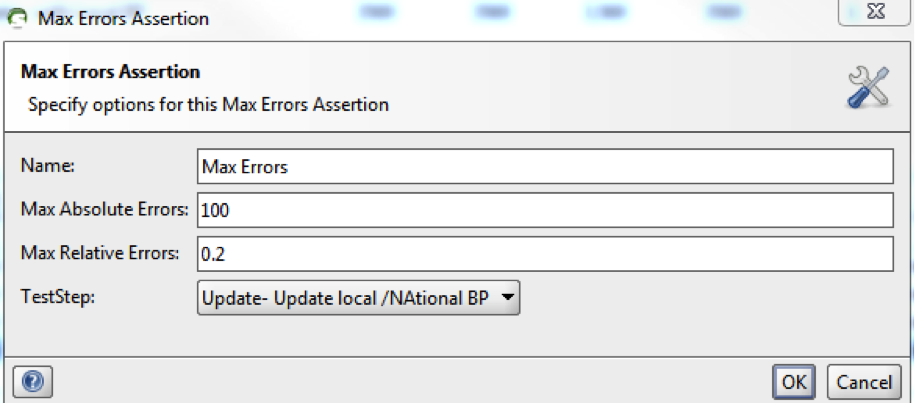
[](http://www.3pillarglobal.com/wp-content/uploads/2016/02/load_testing_3.png)

Load Test Assertions: Assertions measure the actual values against the defined values for the test step and log the number of assertions errors under “err”. We can add assertions to the test steps by right clicking on the step and selecting “Add Assertions”. We have to choose the assertion type while adding the assertion from the given list. The following are the assertion types:

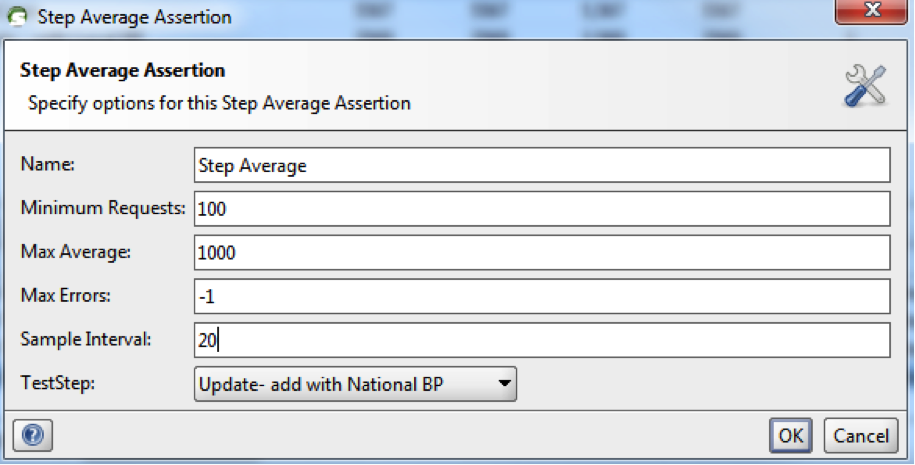
1. Step TPS: This assertion is used to verify the Transaction per second of the test step. When you select this type, a model window opens for this assertion and has the following options—

* Name – Define the assertion name.
* Minimum Requests – This feature lets you define the minimum number of requests that you want to execute before the assertion has to be applied. It’s useful when you start the load test and don’t want to assert some initial requests.
* Minimum TPS – The desired value of the TPS. It will log an assertion error if the actual value is less than this.
* Max Errors – This is the value by which test should be cancelled. “-1” means that there is no such number.
* Test Step – It shows the entire test step defined under the test case and lets you select one on which you have to apply the assertion. It also has the value “Any,” which lets you generalize this assertion for all of the steps, and “Total,” which will assert the total of all the steps against the set value.  
  [](http://www.3pillarglobal.com/wp-content/uploads/2016/02/load_testing_4.png)

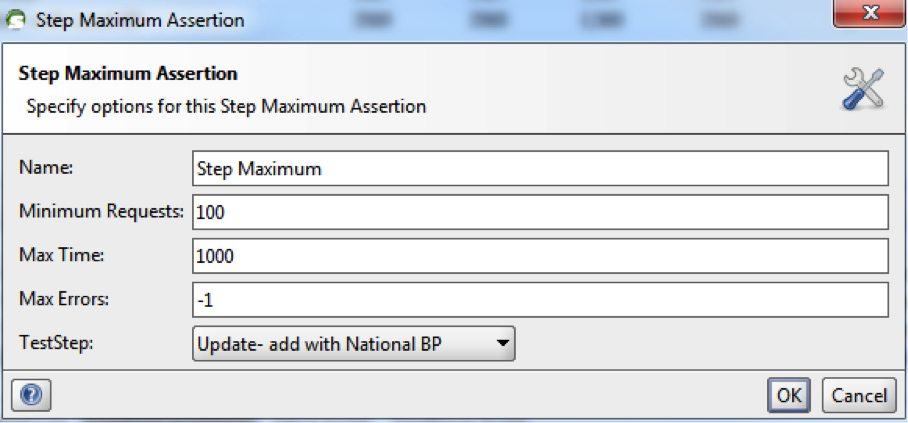
2. Max Errors: This assertion is used to verify the total errors that the test step should not exceed the specified value. When you select this type, a model window opens for this assertion with the following options (skipping the common options mentioned above) —

* Max Absolute Errors – This is the absolute maximum numbers of errors allowed for the test step.
* Max Relative Errors – This can have value between 0 and 1. If the value is set to “0.4,” that means that, at most, 40% of the run of the specified test step should result in error.  
  [](http://www.3pillarglobal.com/wp-content/uploads/2016/02/load_testing_5.png)

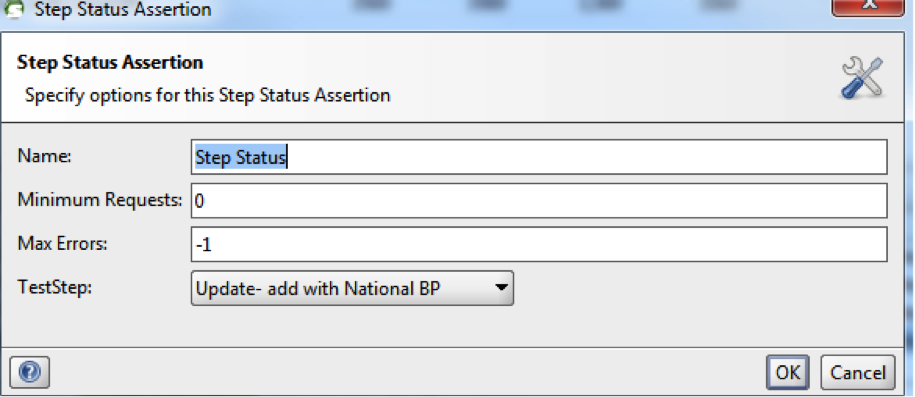
3. Step Average: This assertion is used to verify that the averages of the test step should not exceed the specified value. When you select this type, a model window opens for this assertion with the following options (skipping the common options mentioned above) —

* Max Average – This checks that if the test step average is greater than the defined value then it should log as assert error.
* Sample Interval – The step count used to calculate the average.  
  [](http://www.3pillarglobal.com/wp-content/uploads/2016/02/load_testing_6.png)

4. Step Maximum: This assertion is used to verify that the time taken by the test step should not exceed the specified value. When you select this type, a model window opens for this assertion having the following options (skipping the common options mentioned above) —

* Max Time – The time that the test step should not exceed or else it should log it as an assert error.  
  [](http://www.3pillarglobal.com/wp-content/uploads/2016/02/load_testing_7.png)

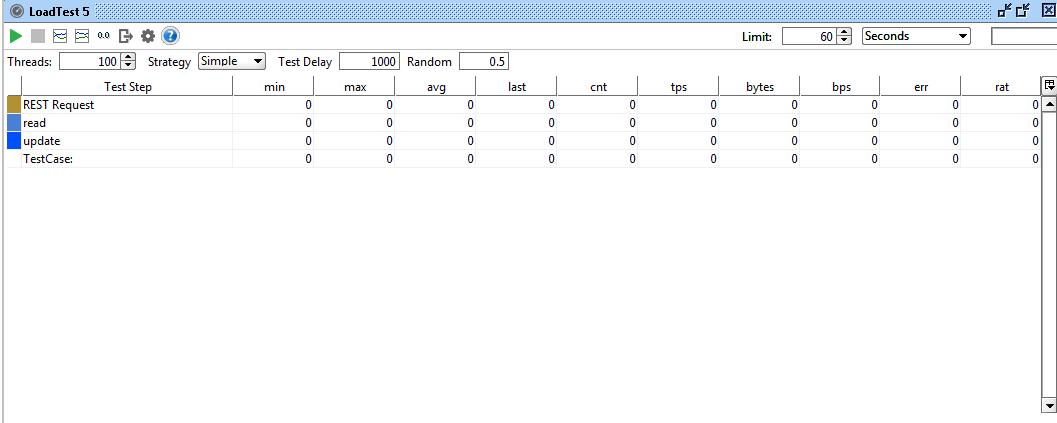
5. Step Status: This assertion is used to verify that the test step is not failed with an error. When you select this type, a model window opens for this assertion with the following options (skipping the common options mentioned above) —

[](http://www.3pillarglobal.com/wp-content/uploads/2016/02/load_testing_8.png)

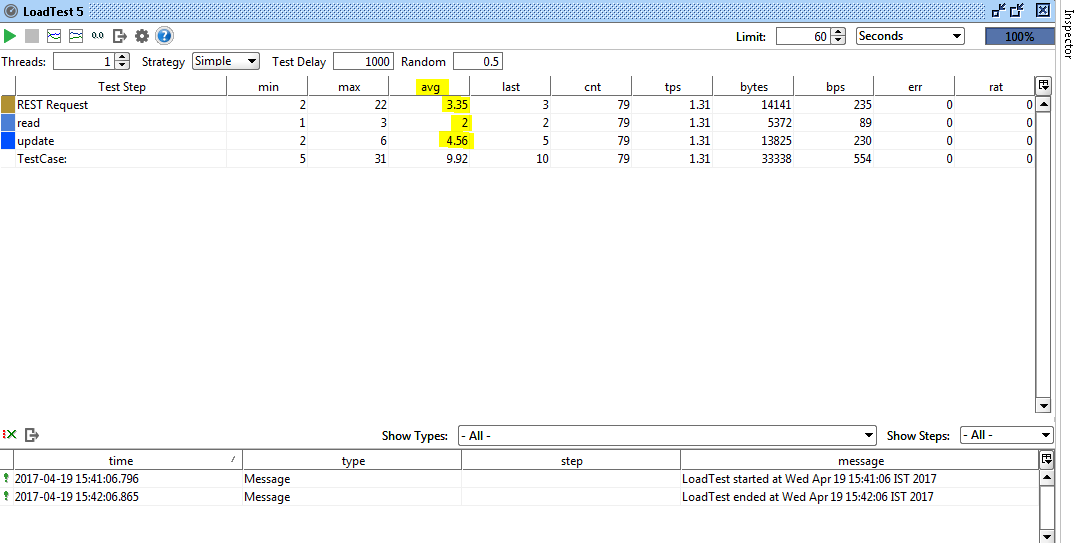
There are Log sub windows in this window in which you can see the logs.

Real time Sample

The above sample we have created test case for only the create ,we have to create test case for all the methods in the same way,and perform load test for all

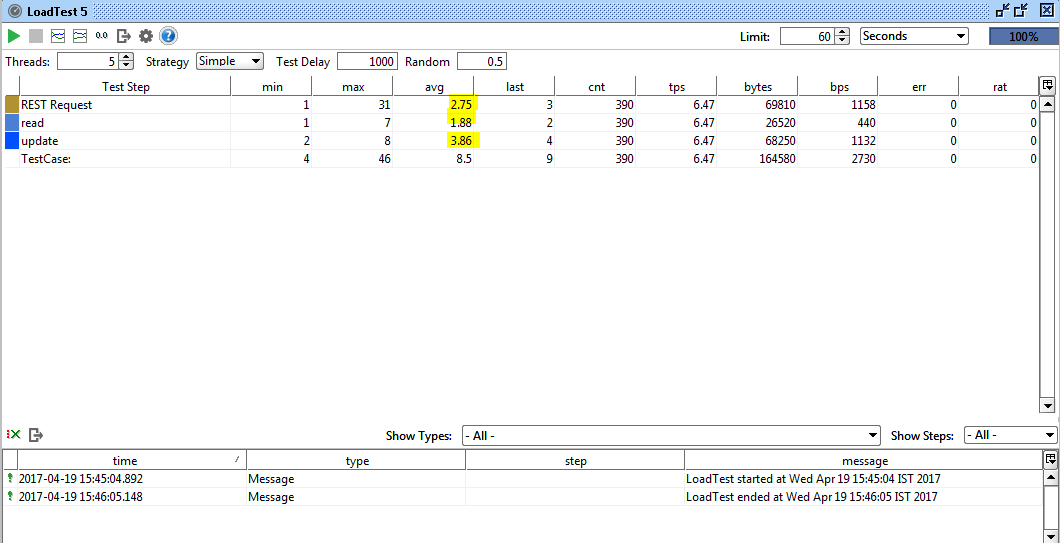


Step 1: Execute with 1 thread. Here is the result:



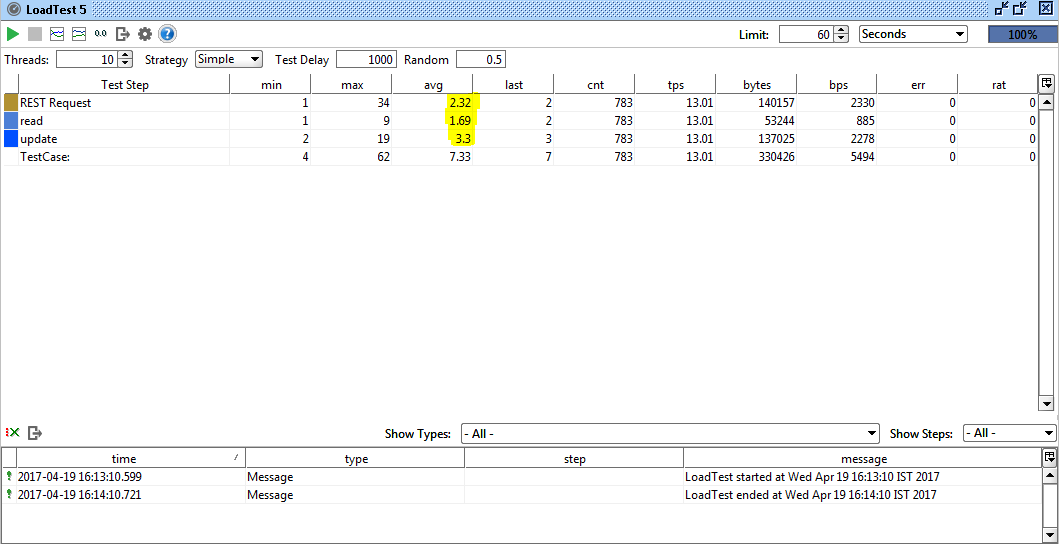
So we can see that create took an average of 3.35 seconds and update took around 4.56 seconds, to read it took around 2 seconds.This looks OK with 1 user load and with the AUT running on the VM, but this is not the actual load test environment.

Step 2: Execute with 5 threads. Here is the result:



we can see that create took an average of 3 seconds and update took around 2 seconds, to read it took around 4 seconds.

Step 3: Execute with 10 threads. Here is the result:



Now the average time to create has gone to around 2 seconds and for update its 3 seconds,read took 2 seconds. Also notice that the max time for create and update has gone up to 34,9 and 19 seconds respectively. Now we can see that even for 10 users, the time is very high as compared to 1 and 5 users.

We can identify these kinds of issues earlier with baseline test, as shown above, which really helps developers to fix the major performance issues at this stage itself.

References:

<https://www.3pillarglobal.com/insights/load-testing-with-soapui-2>

<https://www.soapui.org/functional-testing/properties/working-with-properties.html>

<http://www.softwaretestingclass.com/simple-guide-to-create-load-testing-using-soapui/>