Session Management in cluster using Spring Session with JDBC DataStore

In web applications, user session management is very crucial for managing user state. In this article, we are going to learn about what are the approaches we have been following to manage user sessions in a clustered environment and how we can use [**Spring Session**](https://projects.spring.io/spring-session/) to implement it in a much simpler and more scalable way.

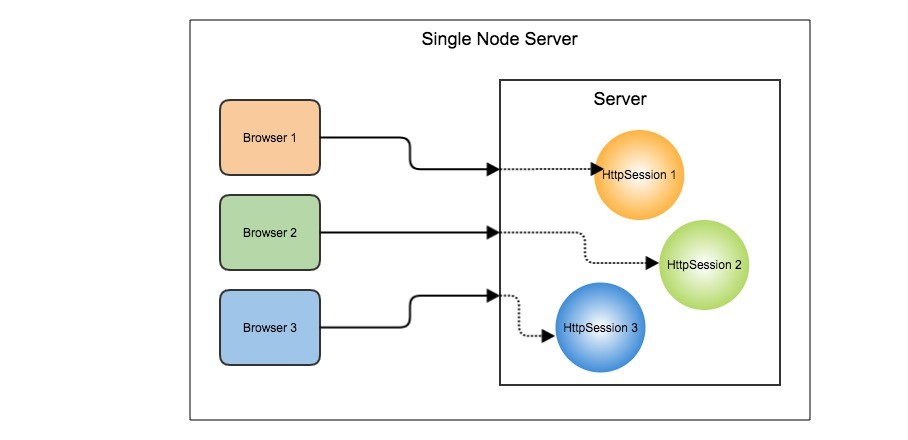
Typically in production environments, we will have multiple server nodes with a load balancer in front of them and all the client traffic will be coming through the load balancer to one of the server nodes. So we need some mechanism to make the user session data available to each client in a clustered environment.  
Traditionally we have been using the following techniques to manage sessions:

1. Single Node Server
2. Multi-Node Server with Load Balancer and Sticky sessions
3. Multi-Node Server with Load Balancer and Session Replication
4. Multi-Node Server with Load Balancer and Session Data in a Persistent DataStore

Let us briefly look at these approaches.

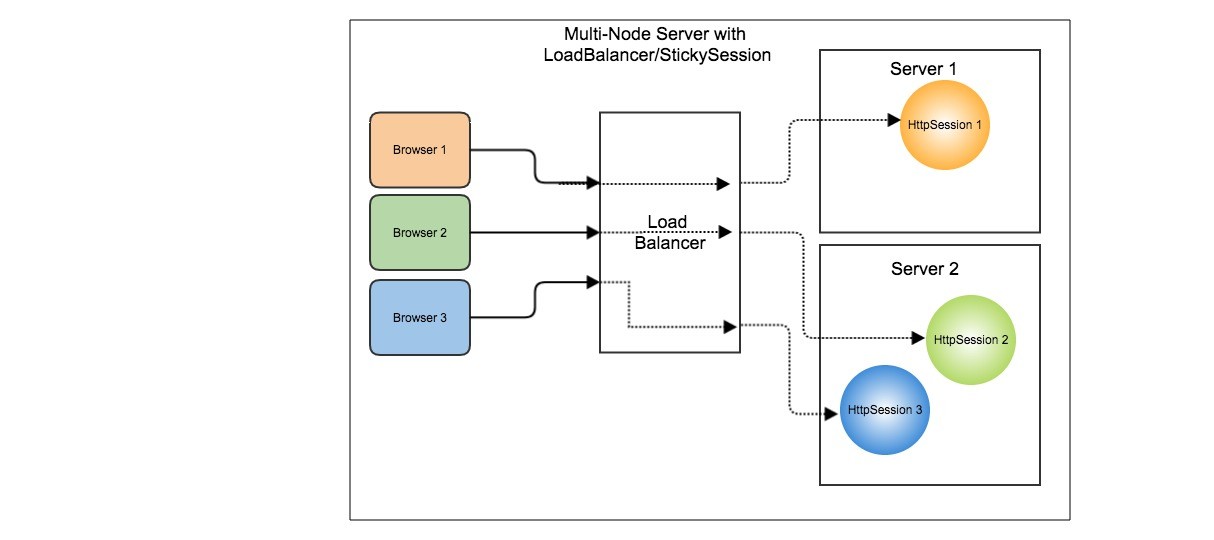
1. Single Node Server

If your application is not a critical service to your business, there won’t be too many users concurrently and some downtime is accepted then we can have Single Node Server deployment as shown below:

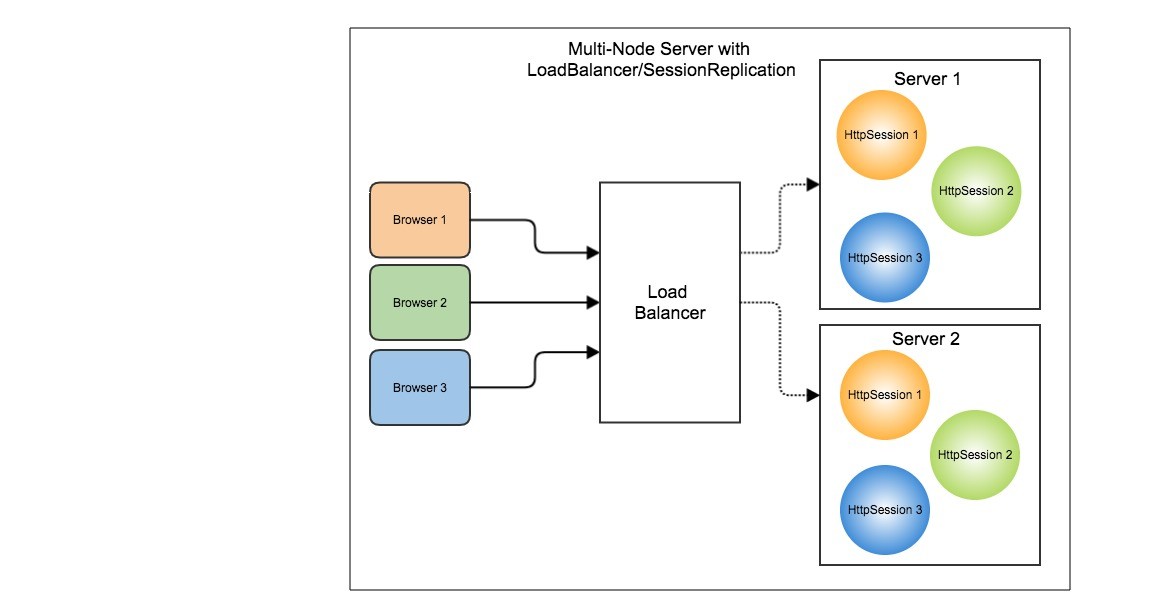
  
In this model, for each browser client, a session object is created on the server (**HttpSession** in case of Java) and SESSION\_ID will be set as a cookie on the browser to identify the session object. But this Single Server Node deployment is not acceptable for most of the applications because if the server goes down the service will be down altogether.

2. Multi-Node Server with Sticky Sessions

In order to make our application highly available and cater more number of users, we can have multiple server nodes behind a load balancer. In the Sticky Sessions approach, we configure our load balancer to route all the requests from the same client to the same node.

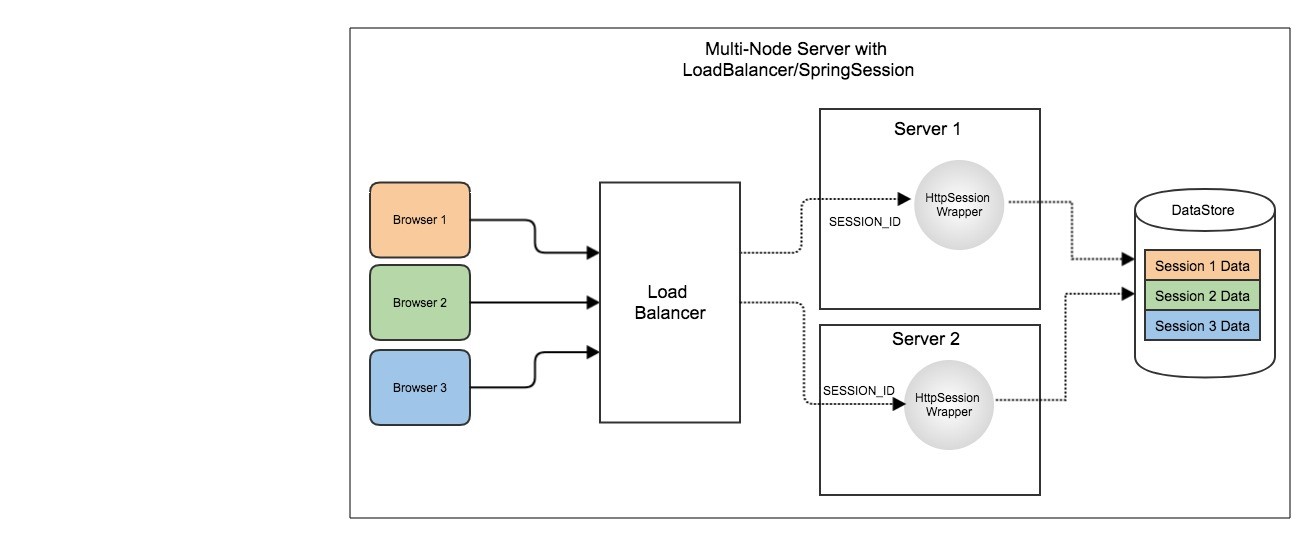
  
In this model, the user session will be created on any one of the server node and all the further requests from that client will be sent to that same node. But the problem with this approach is if a server node goes down then all the user sessions on that server is gone.

3. Multi-Node Server with Session Replication

In this model, the user session data will be replicated on all the server nodes so that any request can be routed to any server node. Even if one node goes down the client request can be served by another node.  


But the Session Replication requires better hardware support and involves some server specific configuration.

4. Multi-Node Server with Session Data in a Persistent DataStore

In this model, the user session data will not be held in server’s memory, instead, it will be persisted into a data store and associate it with SESSION\_ID.  


This solution will be server independent but we may need to write custom code to transparently store the session data in a Persistent datastore whenever a user adds some information to his/her session.

This is where Spring Session comes into the picture.

Spring Session

[Spring Session](https://projects.spring.io/spring-session/) is an implementation of approach 4, which is Storing session data in a persistent datastore. Spring Session supports multiple datastores like RDBMS, Redis, HazelCast, MongoDB etc to transparently save use session data. As usual, using Spring Session with Spring Boot is as simple as adding a dependency and configuring few properties.  
Let us see how we can use Spring Session with JDBC backend store in a Spring Boot application.

*https://github.com/sivaprasadreddy/spring-session-samples*

Step 1: Create Spring Boot application

Create a SpringBoot application using the latest version (it is 2.0.0.RC1 as of writing) with **Web**, **Thymeleaf**, **JPA**, **H2**, **Session** starters.  
By default Session starter will add ***org.springframework.session:spring-session-core dependency***, let us change it to ***spring-session-*jdbc** as we are going to use JDBC backend.

<dependency>

<groupId>org.springframework.session</groupId>

<artifactId>spring-session-jdbc</artifactId>

</dependency>

Step 2: Configure Spring Session properties

We can configure type of Spring Session backend data-store using **spring.session.store-type** property in *application.properties*.

|  |  |
| --- | --- |
| 1 | spring.session.store-type=jdbc |

As we are using the H2 In-Memory database, Spring Session creates the following tables required to store session data automatically from the script ***spring-session-*jdbc*-2.0.1.RELEASE.jar!/org/springframework/session/jdbc/schema-h2.sql***.

CREATE TABLE SPRING\_SESSION (

PRIMARY\_ID CHAR(36) NOT NULL,

SESSION\_ID CHAR(36) NOT NULL,

CREATION\_TIME BIGINT NOT NULL,

LAST\_ACCESS\_TIME BIGINT NOT NULL,

MAX\_INACTIVE\_INTERVAL INT NOT NULL,

EXPIRY\_TIME BIGINT NOT NULL,

PRINCIPAL\_NAME VARCHAR(100),

CONSTRAINT SPRING\_SESSION\_PK PRIMARY KEY (PRIMARY\_ID)

);

CREATE UNIQUE INDEX SPRING\_SESSION\_IX1 ON SPRING\_SESSION (SESSION\_ID);

CREATE INDEX SPRING\_SESSION\_IX2 ON SPRING\_SESSION (EXPIRY\_TIME);

CREATE INDEX SPRING\_SESSION\_IX3 ON SPRING\_SESSION (PRINCIPAL\_NAME);

CREATE TABLE SPRING\_SESSION\_ATTRIBUTES (

SESSION\_PRIMARY\_ID CHAR(36) NOT NULL,

ATTRIBUTE\_NAME VARCHAR(200) NOT NULL,

ATTRIBUTE\_BYTES LONGVARBINARY NOT NULL,

CONSTRAINT SPRING\_SESSION\_ATTRIBUTES\_PK PRIMARY KEY (SESSION\_PRIMARY\_ID, ATTRIBUTE\_NAME),

CONSTRAINT SPRING\_SESSION\_ATTRIBUTES\_FK FOREIGN KEY (SESSION\_PRIMARY\_ID) REFERENCES SPRING\_SESSION(PRIMARY\_ID) ON DELETE CASCADE

);

CREATE INDEX SPRING\_SESSION\_ATTRIBUTES\_IX1 ON SPRING\_SESSION\_ATTRIBUTES (SESSION\_PRIMARY\_ID);

But if we are going to use other RDBMS like MySQL we can configure as follows:

Add MySQL maven dependency.

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

Configure datasource properties for MySQL:

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/demo

spring.datasource.username=root

spring.datasource.password=admin

Enable Spring Session tables creation using **spring.session.jdbc.initialize-schema** property.

spring.session.jdbc.initialize-schema=always

With this property, Spring Session will try to create tables using the script **“classpath:org/springframework/session/jdbc/schema-@@platform@@.sql”**, so in our case, it will use **schema-mysql.sql**.

Step 3: Add data to HttpSession

Now create a simple form in src/main/resources/templates/index.html.

<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title>Spring Session + JDBC Demo</title>

</head>

<body>

<div>

<form th:action="@{/messages}" method="post">

<textarea name="msg" cols="40" rows="4"></textarea>

<input type="submit" value="Save"/>

</form>

</div>

<div>

<h2>Messages</h2>

<ul th:each="m : ${messages}">

<li th:text="${m}">msg</li>

</ul>

</div>

</body>

</html>

Let us implement a Controller to add messages to **HttpSession** and display them.

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestParam;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpSession;

import java.util.\*;

@Controller

public class MessagesController

{

@GetMapping("/")

public String index(Model model, HttpSession session) {

List<String> msgs = (List<String>) session.getAttribute("MY\_MESSAGES");

if(msgs == null) {

msgs = new ArrayList<>();

}

model.addAttribute("messages", msgs);

return "index";

}

@PostMapping("/messages")

public String saveMessage(@RequestParam("msg") String msg, HttpServletRequest request)

{

List<String> msgs = (List<String>) request.getSession().getAttribute("MY\_MESSAGES");

if(msgs == null) {

msgs = new ArrayList<>();

request.getSession().setAttribute("MY\_MESSAGES", msgs);

}

msgs.add(msg);

return "redirect:/";

}

}

Now you can start the application and add some messages to HttpSession and you can see the rows in **SPRING\_SESSION**, **SPRING\_SESSION\_ATTRIBUTES** tables. By default, Spring Session converts the objects that we are trying to add to HttpSession into ByteArray and store it in the table.

Spring Session with Spring Security

**Spring Session** seamlessly integrates with **Spring Security** because of SpringBoot’s auto-configuration.  
Let us add Spring Security to our application.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

Add a default user credentials in **application.properties** as follows:

spring.security.user.name=admin

spring.security.user.password=secret

Now if you try to access http://localhost:8080/ you will be redirected to auto-generated Login page.  
Once you login and see the data in **SPRING\_SESSION** table you can see that login username is stored in **PRINCIPAL\_NAME** column.

How does Spring Session work?

Spring Session provides implementations for **HttpServletRequest** and **HttpSession** which are **SessionRepositoryRequestWrapper** and **HttpSessionWrapper**. Spring Session provides **SessionRepositoryFilter** to intercept all the requests and wrap the **HttpServletRequest** in **SessionRepositoryRequestWrapper**.

In **SessionRepositoryRequestWrapper.getSession(boolean)** is overriden to return **HttpSessionWrapper**object instead of default server implementation of **HttpSession**. The **HttpSessionWrapper** uses **SessionRepository** to persist session information in a datastore.

The SessionRepository interface has various methods to manage sessions.

public interface SessionRepository<S extends Session>

{

S createSession();

void save(S session);

S findById(String id);

void deleteById(String id);

}

This SessionRepository interface is implemented by various classes based on the type of backend we are using. In our case, we are using **JdbcOperationsSessionRepository** provided by **spring-session-jdbc**.

Conclusion

As you might have already observed we can manage user sessions effectively by using Spring Session with very minimal configuration because of Spring Boot auto-configuration. If for any reason we want to change the backend from JDBC to Redis or Hazelcast etc it is only simple configuration change as we are not directly depending on any Spring Session classes.