Spring Session

Table of Contents

Spring Session provides an API and implementations for managing a user’s session information.

1. Introduction

Spring Session provides an API and implementations for managing a user's session information. It also provides transparent integration with:

- HttpSession - allows replacing the HttpSession in an application container (i.e. Tomcat) neutral way. Additional features include:
  - **Clustered Sessions** - Spring Session makes it trivial to support clustered sessions without being tied to an application container specific solution.
  - **Multiple Browser Sessions** - Spring Session supports managing multiple users' sessions in a single browser instance (i.e. multiple authenticated accounts similar to Google).
  - **RESTful APIs** - Spring Session allows providing session ids in headers to work with RESTful APIs
- WebSocket - provides the ability to keep the `HttpSession` alive when receiving WebSocket messages

2. What's New in 1.3

Below are the highlights of what is new in Spring Session 1.3. You can find a complete list of what's new by referring to the changelogs of 1.3.0.M1, 1.3.0.M2, 1.3.0.RC1, and 1.3.0.RELEASE.

- First class support for Hazelcast
- First class support for Spring Security's concurrent session management
- Added OrientDB Community Extension
- GenericJackson2JsonRedisSerializer sample with Spring Security's new Jackson Support
- Guides now use Lettuce
- `spring.session.cleanup.cron.expression` can be used to override the cleanup task's cron expression
- Lots of performance improvements and bug fixes

3. Samples and Guides (Start Here)

If you are looking to get started with Spring Session, the best place to start is our Sample Applications.

Table 3.1. Sample Applications using Spring Boot

Source	Description	Guide
HttpSession with Redis	Demonstrates how to use Spring Session to replace the `HttpSession` with Redis.	HttpSession with Redis Guide
HttpSession with JDBC	Demonstrates how to use Spring Session to replace the `HttpSession` with a relational database store.	HttpSession with JDBC Guide
Find by Username	Demonstrates how to use Spring Session to find sessions by username.	Find by Username Guide
WebSockets	Demonstrates how to use Spring Session with WebSockets.	WebSockets Guide
HttpSession with Redis JSON serialization	Demonstrates how to use Spring Session to replace the `HttpSession` with Redis using JSON serialization.	TBD

Table 3.2. Sample Applications using Spring Java based configuration

Source	Description	Guide
HttpSession with Redis	Demonstrates how to use Spring Session to replace the `HttpSession` with Redis.	HttpSession with Redis Guide
HttpSession with JDBC	Demonstrates how to use Spring Session to replace the `HttpSession` with a relational database store.	HttpSession with JDBC Guide
HttpSession with Hazelcast	Demonstrates how to use Spring Session to replace the `HttpSession` with Hazelcast.	HttpSession with Hazelcast Guide
Custom Cookie	Demonstrates how to use Spring Session and customize the cookie.	Custom Cookie Guide
Spring Security	Demonstrates how to use Spring Session with an existing Spring Security application.	Spring Security Guide

Source	Description	Guide
REST	Demonstrates how to use Spring Session in a REST application to support authenticating with a header.	REST Guide
Multiple Users	Demonstrates how to use Spring Session to manage multiple simultaneous browser sessions (i.e Google Accounts).	Multiple Users Guide

Table 3.3. Sample Applications using Spring XML based configuration

Source	Description	Guide
HttpSession with Redis	Demonstrates how to use Spring Session to replace the `HttpSession` with a Redis store.	HttpSession with Redis Guide
HttpSession with JDBC	Demonstrates how to use Spring Session to replace the `HttpSession` with a relational database store.	HttpSession with JDBC Guide

Table 3.4. Misc sample Applications

Source	Description	Guide
Grails 3	Demonstrates how to use Spring Session with Grails 3.	Grails 3 Guide
Hazelcast	Demonstrates how to use Spring Session with Hazelcast in a Java EE application.	TBD

4. HttpSession Integration

Spring Session provides transparent integration with `HttpSession`. This means that developers can switch the `HttpSession` implementation out with an implementation that is backed by Spring Session.

4.1 Why Spring Session & HttpSession?

We have already mentioned that Spring Session provides transparent integration with `HttpSession`, but what benefits do we get out of this?

- **Clustered Sessions** - Spring Session makes it trivial to support clustered sessions without being tied to an application container specific solution.
- **Multiple Browser Sessions** - Spring Session supports managing multiple users' sessions in a single browser instance (i.e. multiple authenticated accounts similar to Google).
- **RESTful APIs** - Spring Session allows providing session ids in headers to work with RESTful APIs

4.2 HttpSession with Redis

Using Spring Session with `HttpSession` is enabled by adding a Servlet Filter before anything that uses the `HttpSession`. You can choose from enabling this using either:

- Java Based Configuration
- XML Based Configuration

Redis Java Based Configuration

This section describes how to use Redis to back `HttpSession` using Java based configuration.

> **Note**
>
> The HttpSession Sample provides a working sample on how to integrate Spring Session and `HttpSession` using Java configuration. You can read the basic steps for integration below, but you are encouraged to follow along with the detailed HttpSession Guide when integrating with your own application.

Spring Java Configuration

After adding the required dependencies, we can create our Spring configuration. The Spring configuration is responsible for creating a Servlet Filter that replaces the `HttpSession` implementation with an implementation backed by Spring Session. Add the following Spring Configuration:

```
@EnableRedisHttpSession ❶
public class Config {

 @Bean
 public LettuceConnectionFactory connectionFactory() {
  return new LettuceConnectionFactory(); ❷
 }
}
```

❶ The `@EnableRedisHttpSession` annotation creates a Spring Bean with the name of `springSessionRepositoryFilter` that implements Filter. The filter is what is in charge of

replacing the `HttpSession` implementation to be backed by Spring Session. In this instance Spring Session is backed by Redis.

❷ We create a `RedisConnectionFactory` that connects Spring Session to the Redis Server. We configure the connection to connect to localhost on the default port (6379) For more information on configuring Spring Data Redis, refer to the reference documentation.

Java Servlet Container Initialization

Our Spring Configuration created a Spring Bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, Spring needs to load our `Config` class. Last we need to ensure that our Servlet Container (i.e. Tomcat) uses our `springSessionRepositoryFilter` for every request. Fortunately, Spring Session provides a utility class named `AbstractHttpSessionApplicationInitializer` both of these steps extremely easy. You can find an example below:

src/main/java/sample/Initializer.java.

```
public class Initializer extends AbstractHttpSessionApplicationInitializer { ❶

 public Initializer() {
  super(Config.class); ❷
 }
}
```

> **Note**
>
> The name of our class (Initializer) does not matter. What is important is that we extend `AbstractHttpSessionApplicationInitializer`.

❶ The first step is to extend `AbstractHttpSessionApplicationInitializer`. This ensures that the Spring Bean by the name `springSessionRepositoryFilter` is registered with our Servlet Container for every request.

❷ `AbstractHttpSessionApplicationInitializer` also provides a mechanism to easily ensure Spring loads our `Config`.

Redis XML Based Configuration

This section describes how to use Redis to back `HttpSession` using XML based configuration.

> **Note**
>
> The HttpSession XML Sample provides a working sample on how to integrate Spring Session and `HttpSession` using XML configuration. You can read the basic steps for integration below, but you are encouraged to follow along with the detailed HttpSession XML Guide when integrating with your own application.

Spring XML Configuration

After adding the required dependencies, we can create our Spring configuration. The Spring configuration is responsible for creating a Servlet Filter that replaces the `HttpSession` implementation with an implementation backed by Spring Session. Add the following Spring Configuration:

src/main/webapp/WEB-INF/spring/session.xml.

```
❶
<context:annotation-config/>
<bean class="org.springframework.session.data.redis.config.annotation.web.http.RedisHttpSessionConfiguration"/
>

❷
<bean class="org.springframework.data.redis.connection.lettuce.LettuceConnectionFactory"/>
```

❶ We use the combination of `<context:annotation-config/>` and `RedisHttpSessionConfiguration` because Spring Session does not yet provide XML Namespace support (see gh-104). This creates a Spring Bean with the name of `springSessionRepositoryFilter` that implements Filter. The filter is what is in charge of replacing the `HttpSession` implementation to be backed by Spring Session. In this instance Spring Session is backed by Redis.

❷ We create a `RedisConnectionFactory` that connects Spring Session to the Redis Server. We configure the connection to connect to localhost on the default port (6379) For more information on configuring Spring Data Redis, refer to the reference documentation.

XML Servlet Container Initialization

Our Spring Configuration created a Spring Bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, we need to instruct Spring to load our `session.xml` configuration. We do this with the following configuration:

src/main/webapp/WEB-INF/web.xml.

```
<context-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>
  /WEB-INF/spring/*.xml
 </param-value>
</context-param>
<listener>
 <listener-class>
  org.springframework.web.context.ContextLoaderListener
 </listener-class>
</listener>
```

The ContextLoaderListener reads the contextConfigLocation and picks up our session.xml configuration.

Last we need to ensure that our Servlet Container (i.e. Tomcat) uses our `springSessionRepositoryFilter` for every request. The following snippet performs this last step for us:

src/main/webapp/WEB-INF/web.xml.

```
<filter>
 <filter-name>springSessionRepositoryFilter</filter-name>
 <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
 <filter-name>springSessionRepositoryFilter</filter-name>
 <url-pattern>/*</url-pattern>
 <dispatcher>REQUEST</dispatcher>
 <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

The DelegatingFilterProxy will look up a Bean by the name of `springSessionRepositoryFilter` and cast it to a `Filter`. For every request that `DelegatingFilterProxy` is invoked, the `springSessionRepositoryFilter` will be invoked.

4.3 HttpSession with JDBC

Using Spring Session with `HttpSession` is enabled by adding a Servlet Filter before anything that uses the `HttpSession`. You can choose from enabling this using either:

- Java Based Configuration
- XML Based Configuration
- Spring Boot Based Configuration

JDBC Java Based Configuration

This section describes how to use a relational database to back `HttpSession` using Java based configuration.

> **Note**
>
> The HttpSession JDBC Sample provides a working sample on how to integrate Spring Session and `HttpSession` using Java configuration. You can read the basic steps for integration below, but you are encouraged to follow along with the detailed HttpSession JDBC Guide when integrating with your own application.

Spring Java Configuration

After adding the required dependencies, we can create our Spring configuration. The Spring configuration is responsible for creating a Servlet Filter that replaces the `HttpSession` implementation with an implementation backed by Spring Session. Add the following Spring Configuration:

```
@EnableJdbcHttpSession ❶
public class Config {

 @Bean
 public EmbeddedDatabase dataSource() {
  return new EmbeddedDatabaseBuilder() ❷
    .setType(EmbeddedDatabaseType.H2)
    .addScript("org/springframework/session/jdbc/schema-h2.sql").build();
 }

 @Bean
 public PlatformTransactionManager transactionManager(DataSource dataSource) {
  return new DataSourceTransactionManager(dataSource); ❸
 }

}
```

❶ The `@EnableJdbcHttpSession` annotation creates a Spring Bean with the name of `springSessionRepositoryFilter` that implements Filter. The filter is what is in charge of replacing the `HttpSession` implementation to be backed by Spring Session. In this instance Spring Session is backed by a relational database.
❷ We create a `dataSource` that connects Spring Session to an embedded instance of H2 database. We configure the H2 database to create database tables using the SQL script which is included in Spring Session.
❸ We create a `transactionManager` that manages transactions for previously configured `dataSource`.

For additional information on how to configure data access related concerns, please refer to the Spring Framework Reference Documentation.

Java Servlet Container Initialization

Our Spring Configuration created a Spring Bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, Spring needs to load our `Config` class. Last we need to ensure that our Servlet Container (i.e. Tomcat) uses our `springSessionRepositoryFilter` for every request. Fortunately, Spring Session provides a utility class named `AbstractHttpSessionApplicationInitializer` both of these steps extremely easy. You can find an example below:

src/main/java/sample/Initializer.java.

```
public class Initializer extends AbstractHttpSessionApplicationInitializer { ❶

 public Initializer() {
  super(Config.class); ❷
 }
}
```

> **Note**
>
> The name of our class (Initializer) does not matter. What is important is that we extend `AbstractHttpSessionApplicationInitializer`.

❶ The first step is to extend `AbstractHttpSessionApplicationInitializer`. This ensures that the Spring Bean by the name `springSessionRepositoryFilter` is registered with our Servlet Container for every request.
❷ `AbstractHttpSessionApplicationInitializer` also provides a mechanism to easily ensure Spring loads our `Config`.

JDBC XML Based Configuration

This section describes how to use a relational database to back `HttpSession` using XML based configuration.

> **Note**
>
> The HttpSession JDBC XML Sample provides a working sample on how to integrate Spring Session and `HttpSession` using XML configuration. You can read the basic steps for integration

> below, but you are encouraged to follow along with the detailed HttpSession JDBC XML Guide when integrating with your own application.

Spring XML Configuration

After adding the required dependencies, we can create our Spring configuration. The Spring configuration is responsible for creating a Servlet Filter that replaces the `HttpSession` implementation with an implementation backed by Spring Session. Add the following Spring Configuration:

src/main/webapp/WEB-INF/spring/session.xml.

```
❶
<context:annotation-config/>
<bean class="org.springframework.session.jdbc.config.annotation.web.http.JdbcHttpSessionConfiguration"/>

❷
<jdbc:embedded-database id="dataSource" database-name="testdb" type="H2">
 <jdbc:script location="classpath:org/springframework/session/jdbc/schema-h2.sql"/>
</jdbc:embedded-database>

❸
<bean class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
 <constructor-arg ref="dataSource"/>
</bean>
```

❶ We use the combination of `<context:annotation-config/>` and `JdbcHttpSessionConfiguration` because Spring Session does not yet provide XML Namespace support (see gh-104). This creates a Spring Bean with the name of `springSessionRepositoryFilter` that implements Filter. The filter is what is in charge of replacing the `HttpSession` implementation to be backed by Spring Session. In this instance Spring Session is backed by a relational database.

❷ We create a `dataSource` that connects Spring Session to an embedded instance of H2 database. We configure the H2 database to create database tables using the SQL script which is included in Spring Session.

❸ We create a `transactionManager` that manages transactions for previously configured `dataSource`.

For additional information on how to configure data access related concerns, please refer to the Spring Framework Reference Documentation.

XML Servlet Container Initialization

Our Spring Configuration created a Spring Bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, we need to instruct Spring to load our `session.xml` configuration. We do this with the following configuration:

src/main/webapp/WEB-INF/web.xml.

```
<context-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>
  /WEB-INF/spring/*.xml
 </param-value>
</context-param>
<listener>
 <listener-class>
  org.springframework.web.context.ContextLoaderListener
 </listener-class>
</listener>
```

The ContextLoaderListener reads the contextConfigLocation and picks up our session.xml configuration.

Last we need to ensure that our Servlet Container (i.e. Tomcat) uses our `springSessionRepositoryFilter` for every request. The following snippet performs this last step for us:

src/main/webapp/WEB-INF/web.xml.

```
<filter>
 <filter-name>springSessionRepositoryFilter</filter-name>
 <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
 <filter-name>springSessionRepositoryFilter</filter-name>
 <url-pattern>/*</url-pattern>
 <dispatcher>REQUEST</dispatcher>
 <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

The DelegatingFilterProxy will look up a Bean by the name of `springSessionRepositoryFilter` and cast it to a `Filter`. For every request that `DelegatingFilterProxy` is invoked, the `springSessionRepositoryFilter` will be invoked.

JDBC Spring Boot Based Configuration

This section describes how to use a relational database to back `HttpSession` when using Spring Boot.

> **Note**
>
> The HttpSession JDBC Spring Boot Sample provides a working sample on how to integrate Spring Session and `HttpSession` using Spring Boot. You can read the basic steps for integration below, but you are encouraged to follow along with the detailed HttpSession JDBC Spring Boot Guide when integrating with your own application.

Spring Boot Configuration

After adding the required dependencies, we can create our Spring Boot configuration. Thanks to first-class auto configuration support, setting up Spring Session backed by a relational database is as simple as adding a single configuration property to your `application.properties`:

src/main/resources/application.properties.

```
spring.session.store-type=jdbc
```

Under the hood, Spring Boot will apply configuration that is equivalent to manually adding `@EnableJdbcHttpSession` annotation. This creates a Spring Bean with the name of

`springSessionRepositoryFilter` that implements Filter. The filter is what is in charge of replacing the `HttpSession` implementation to be backed by Spring Session.

Further customization is possible using `application.properties`:

src/main/resources/application.properties.

```
server.session.timeout= # Session timeout in seconds.
spring.session.jdbc.initializer.enabled= # Create the required session tables on startup if necessary.
 Enabled automatically if the default table name is set or a custom schema is configured.
spring.session.jdbc.schema=classpath:org/springframework/session/jdbc/schema-@@platform@@.sql # Path to
 the SQL file to use to initialize the database schema.
spring.session.jdbc.table-name=SPRING_SESSION # Name of database table used to store sessions.
```

For more information, refer to Spring Session portion of the Spring Boot documentation.

Configuring the DataSource

Spring Boot automatically creates a `DataSource` that connects Spring Session to an embedded instance of H2 database. In a production environment you need to ensure to update your configuration to point to your relational database. For example, you can include the following in your **application.properties**

src/main/resources/application.properties.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/myapp
spring.datasource.username=myapp
spring.datasource.password=secret
```

For more information, refer to Configure a DataSource portion of the Spring Boot documentation.

Servlet Container Initialization

Our Spring Boot Configuration created a Spring Bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, Spring needs to load our `Config` class. Last we need to ensure that our Servlet Container (i.e. Tomcat) uses our `springSessionRepositoryFilter` for every request. Fortunately, Spring Boot takes care of both of these steps for us.

4.4 HttpSession with Hazelcast

Using Spring Session with `HttpSession` is enabled by adding a Servlet Filter before anything that uses the `HttpSession`.

This section describes how to use Hazelcast to back `HttpSession` using Java based configuration.

> **Note**
>
> The Hazelcast Spring Sample provides a working sample on how to integrate Spring Session and `HttpSession` using Java configuration. You can read the basic steps for integration below, but you are encouraged to follow along with the detailed Hazelcast Spring Guide when integrating with your own application.

Spring Configuration

After adding the required dependencies, we can create our Spring configuration. The Spring configuration is responsible for creating a Servlet Filter that replaces the `HttpSession` implementation with an implementation backed by Spring Session. Add the following Spring Configuration:

```
@EnableHazelcastHttpSession ❶
@Configuration
public class HazelcastHttpSessionConfig {

 @Bean
 public HazelcastInstance hazelcastInstance() {
  MapAttributeConfig attributeConfig = new MapAttributeConfig()
    .setName(HazelcastSessionRepository.PRINCIPAL_NAME_ATTRIBUTE)
    .setExtractor(PrincipalNameExtractor.class.getName());

  Config config = new Config();

  config.getMapConfig("spring:session:sessions") ❷
    .addMapAttributeConfig(attributeConfig)
    .addMapIndexConfig(new MapIndexConfig(
      HazelcastSessionRepository.PRINCIPAL_NAME_ATTRIBUTE, false));

  return Hazelcast.newHazelcastInstance(config); ❸
 }

}
```

❶ The `@EnableHazelcastHttpSession` annotation creates a Spring Bean with the name of `springSessionRepositoryFilter` that implements Filter. The filter is what is in charge of replacing the `HttpSession` implementation to be backed by Spring Session. In this instance Spring Session is backed by Hazelcast.

❷ In order to support retrieval of sessions by principal name index, appropriate `ValueExtractor` needs to be registered. Spring Session provides `PrincipalNameExtractor` for this purpose.

❸ We create a `HazelcastInstance` that connects Spring Session to Hazelcast. By default, an embedded instance of Hazelcast is started and connected to by the application. For more information on configuring Hazelcast, refer to the reference documentation.

Servlet Container Initialization

Our Spring Configuration created a Spring Bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, Spring needs to load our `SessionConfig` class. Since our application is already loading Spring configuration using our `SecurityInitializer` class, we can simply add our `SessionConfig` class to it.

src/main/java/sample/SecurityInitializer.java.

```
public class SecurityInitializer extends AbstractSecurityWebApplicationInitializer {

 public SecurityInitializer() {
  super(SecurityConfig.class, SessionConfig.class);
 }
}
```

Last we need to ensure that our Servlet Container (i.e. Tomcat) uses our `springSessionRepositoryFilter` for every request. It is extremely important that Spring Session's `springSessionRepositoryFilter` is invoked before Spring Security's

`springSecurityFilterChain`. This ensures that the `HttpSession` that Spring Security uses is backed by Spring Session. Fortunately, Spring Session provides a utility class named `AbstractHttpSessionApplicationInitializer` that makes this extremely easy. You can find an example below:

src/main/java/sample/Initializer.java.

```
public class Initializer extends AbstractHttpSessionApplicationInitializer {

}
```

> **Note**
>
> The name of our class (Initializer) does not matter. What is important is that we extend `AbstractHttpSessionApplicationInitializer`.

By extending `AbstractHttpSessionApplicationInitializer` we ensure that the Spring Bean by the name `springSessionRepositoryFilter` is registered with our Servlet Container for every request before Spring Security's `springSecurityFilterChain`.

4.5 How HttpSession Integration Works

Fortunately both `HttpSession` and `HttpServletRequest` (the API for obtaining an `HttpSession`) are both interfaces. This means that we can provide our own implementations for each of these APIs.

> **Note**
>
> This section describes how Spring Session provides transparent integration with `HttpSession`. The intent is so that user's can understand what is happening under the covers. This functionality is already integrated and you do NOT need to implement this logic yourself.

First we create a custom `HttpServletRequest` that returns a custom implementation of `HttpSession`. It looks something like the following:

```
public class SessionRepositoryRequestWrapper extends HttpServletRequestWrapper {

 public SessionRepositoryRequestWrapper(HttpServletRequest original) {
  super(original);
 }

 public HttpSession getSession() {
  return getSession(true);
 }

 public HttpSession getSession(boolean createNew) {
  // create an HttpSession implementation from Spring Session
 }

 // ... other methods delegate to the original HttpServletRequest ...
}
```

Any method that returns an `HttpSession` is overridden. All other methods are implemented by `HttpServletRequestWrapper` and simply delegate to the original `HttpServletRequest` implementation.

We replace the `HttpServletRequest` implementation using a servlet `Filter` called `SessionRepositoryFilter`. The pseudocode can be found below:

```
public class SessionRepositoryFilter implements Filter {

 public doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {
  HttpServletRequest httpRequest = (HttpServletRequest) request;
  SessionRepositoryRequestWrapper customRequest =
   new SessionRepositoryRequestWrapper(httpRequest);

  chain.doFilter(customRequest, response, chain);
 }

 // ...
}
```

By passing in a custom `HttpServletRequest` implementation into the `FilterChain` we ensure that anything invoked after our `Filter` uses the custom `HttpSession` implementation. This highlights why it is important that Spring Session's `SessionRepositoryFilter` must be placed before anything that interacts with the `HttpSession`.

4.6 Multiple HttpSessions in Single Browser

Spring Session has the ability to support multiple sessions in a single browser instance. This provides the ability to support authenticating with multiple users in the same browser instance (i.e. Google Accounts).

> **Note**
>
> The Manage Multiple Users Guide provides a complete working example of managing multiple users in the same browser instance. You can follow the basic steps for integration below, but you are encouraged to follow along with the detailed Manage Multiple Users Guide when integrating with your own application.

Let's take a look at how Spring Session keeps track of multiple sessions.

Managing a Single Session

Spring Session keeps track of the `HttpSession` by adding a value to a cookie named SESSION. For example, the SESSION cookie might have a value of:

```
7e8383a4-082c-4ffe-a4bc-c40fd3363c5e
```

Adding a Session

We can add another session by requesting a URL that contains a special parameter in it. By default the parameter name is **_s**. For example, the following URL would create a new session:

http://localhost:8080/?_s=1

> **Note**
>
> The parameter value does not indicate the actual session id. This is important because we never want to allow the session id to be determined by a client to avoid session fixation attacks. Additionally, we do not want the session id to be leaked since it is sent as a query parameter. Remember sensitive information should only be transmitted as a header or in the body of the request.

Rather than creating the URL ourselves, we can utilize the `HttpSessionManager` to do this for us. We can obtain the `HttpSessionManager` from the `HttpServletRequest` using the following:

src/main/java/sample/UserAccountsFilter.java.

```
HttpSessionManager sessionManager = (HttpSessionManager) httpRequest
  .getAttribute(HttpSessionManager.class.getName());
```

We can now use it to create a URL to add another session.

src/main/java/sample/UserAccountsFilter.java.

```
String addAlias = unauthenticatedAlias == null ? ❶
  sessionManager.getNewSessionAlias(httpRequest)
  : ❷
  unauthenticatedAlias; ❸
String addAccountUrl = sessionManager.encodeURL(contextPath, addAlias); ❹
```

❶ We have an existing variable named `unauthenticatedAlias`. The value is an alias that points to an existing unauthenticated session. If no such session exists, the value is null. This ensures if we have an existing unauthenticated session that we use it instead of creating a new session.

❷ If all of our sessions are already associated to a user, we create a new session alias.

❸ If there is an existing session that is not associated to a user, we use its session alias.

❹ Finally, we create the add account URL. The URL contains a session alias that either points to an existing unauthenticated session or is an alias that is unused thus signaling to create a new session associated to that alias.

Now our SESSION cookie looks something like this:

```
0 7e8383a4-082c-4ffe-a4bc-c40fd3363c5e 1 1d526d4a-c462-45a4-93d9-84a39b6d44ad
```

Such that:

- There is a session with the id **7e8383a4-082c-4ffe-a4bc-c40fd3363c5e**
  - The alias for this session is **0**. For example, if the URL is http://localhost:8080/?_s=0 this alias would be used.
  - This is the default session. This means that if no session alias is specified, then this session is used. For example, if the URL is http://localhost:8080/ this session would be used.
- There is a session with the id **1d526d4a-c462-45a4-93d9-84a39b6d44ad**
  - The alias for this session is **1**. If the session alias is **1**, then this session is used. For example, if the URL is http://localhost:8080/?_s=1 this alias would be used.

Automatic Session Alias Inclusion with encodeURL

The nice thing about specifying the session alias in the URL is that we can have multiple tabs open with different active sessions. The bad thing is that we need to include the session alias in every URL of our application. Fortunately, Spring Session will automatically include the session alias in any URL that passes through HttpServletResponse#encodeURL(java.lang.String)

This means that if you are using standard tag libraries the session alias is automatically included in the URL. For example, if we are currently using the session with the alias of **1**, then the following:

src/main/webapp/index.jsp.

```
  -->
<c:url value="/link.jsp" var="linkUrl"/>
<a id="navLink" href="${linkUrl}">Link</a>
```

will output a link of:

```
<a id="navLink" href="/link.jsp?_s=1">Link</a>
```

4.7 HttpSession & RESTful APIs

Spring Session can work with RESTful APIs by allowing the session to be provided in a header.

> **Note**
>
> The REST Sample provides a working sample on how to use Spring Session in a REST application to support authenticating with a header. You can follow the basic steps for integration below, but you are encouraged to follow along with the detailed REST Guide when integrating with your own application.

Spring Configuration

After adding the required dependencies, we can create our Spring configuration. The Spring configuration is responsible for creating a Servlet Filter that replaces the `HttpSession` implementation with an implementation backed by Spring Session. Add the following Spring Configuration:

```
@Configuration
@EnableRedisHttpSession ❶
public class HttpSessionConfig {

 @Bean
 public LettuceConnectionFactory connectionFactory() {
  return new LettuceConnectionFactory(); ❷
 }

 @Bean
 public HttpSessionStrategy httpSessionStrategy() {
  return new HeaderHttpSessionStrategy(); ❸
 }
}
```

❶ The `@EnableRedisHttpSession` annotation creates a Spring Bean with the name of `springSessionRepositoryFilter` that implements `Filter`. The filter is what is in charge of replacing the `HttpSession` implementation to be backed by Spring Session. In this instance Spring Session is backed by Redis.

❷ We create a `RedisConnectionFactory` that connects Spring Session to the Redis Server. We configure the connection to connect to localhost on the default port (6379) For more information on configuring Spring Data Redis, refer to the reference documentation.

❸ We customize Spring Session's HttpSession integration to use HTTP headers to convey the current session information instead of cookies.

Servlet Container Initialization

Our Spring Configuration created a Spring Bean named `springSessionRepositoryFilter` that implements `Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `HttpSession` with a custom implementation that is backed by Spring Session.

In order for our `Filter` to do its magic, Spring needs to load our `Config` class. We provide the configuration in our Spring `MvcInitializer` as shown below:

src/main/java/sample/mvc/MvcInitializer.java.

```
@Override
protected Class<?>[] getRootConfigClasses() {
 return new Class[] { SecurityConfig.class, HttpSessionConfig.class };
}
```

Last we need to ensure that our Servlet Container (i.e. Tomcat) uses our `springSessionRepositoryFilter` for every request. Fortunately, Spring Session provides a utility class named `AbstractHttpSessionApplicationInitializer` that makes this extremely easy. Simply extend the class with the default constructor as shown below:

src/main/java/sample/Initializer.java.

```
public class Initializer extends AbstractHttpSessionApplicationInitializer {

}
```

> **Note**
>
> The name of our class (Initializer) does not matter. What is important is that we extend `AbstractHttpSessionApplicationInitializer`.

4.8 HttpSessionListener

Spring Session supports `HttpSessionListener` by translating `SessionDestroyedEvent` and `SessionCreatedEvent` into `HttpSessionEvent` by declaring `SessionEventHttpSessionListenerAdapter`. To use this support, you need to:

- Ensure your `SessionRepository` implementation supports and is configured to fire `SessionDestroyedEvent` and `SessionCreatedEvent`.
- Configure `SessionEventHttpSessionListenerAdapter` as a Spring bean.
- Inject every `HttpSessionListener` into the `SessionEventHttpSessionListenerAdapter`

If you are using the configuration support documented in HttpSession with Redis, then all you need to do is register every `HttpSessionListener` as a bean. For example, assume you want to support Spring Security's concurrency control and need to use `HttpSessionEventPublisher` you can simply add `HttpSessionEventPublisher` as a bean. In Java configuration, this might look like:

```
@Configuration
@EnableRedisHttpSession
public class RedisHttpSessionConfig {

 @Bean
 public HttpSessionEventPublisher httpSessionEventPublisher() {
  return new HttpSessionEventPublisher();
 }

 // ...
}
```

In XML configuration, this might look like:

```
<bean class="org.springframework.security.web.session.HttpSessionEventPublisher"/>
```

5. WebSocket Integration

Spring Session provides transparent integration with Spring's WebSocket support.

> **Note**
>
> Spring Session's WebSocket support only works with Spring's WebSocket support. Specifically it does not work with using JSR-356 directly. This is due to the fact that JSR-356 does not have a mechanism for intercepting incoming WebSocket messages.

5.1 Why Spring Session & WebSockets?

So why do we need Spring Session when using WebSockets?

Consider an email application that does much of its work through HTTP requests. However, there is also a chat application embedded within it that works over WebSocket APIs. If a user is actively chatting with someone, we should not timeout the `HttpSession` since this would be pretty poor user experience. However, this is exactly what JSR-356 does.

Another issue is that according to JSR-356 if the `HttpSession` times out any WebSocket that was created with that HttpSession and an authenticated user should be forcibly closed. This means that if we are actively chatting in our application and are not using the HttpSession, then we will also disconnect from our conversation!

5.2 WebSocket Usage

The WebSocket Sample provides a working sample on how to integrate Spring Session with WebSockets. You can follow the basic steps for integration below, but you are encouraged to follow along with the detailed WebSocket Guide when integrating with your own application:

HttpSession Integration

Before using WebSocket integration, you should be sure that you have Chapter 4, *HttpSession Integration* working first.

Spring Configuration

In a typical Spring WebSocket application users would extend `AbstractWebSocketMessageBrokerConfigurer`. For example, the configuration might look something like the following:

```
@Configuration
@EnableScheduling
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {

 public void registerStompEndpoints(StompEndpointRegistry registry) {
  registry.addEndpoint("/messages").withSockJS();
 }

 @Override
 public void configureMessageBroker(MessageBrokerRegistry registry) {
  registry.enableSimpleBroker("/queue/", "/topic/");
  registry.setApplicationDestinationPrefixes("/app");
 }
}
```

We can easily update our configuration to use Spring Session's WebSocket support. For example:

src/main/java/samples/config/WebSocketConfig.java.

```
@Configuration
@EnableScheduling
@EnableWebSocketMessageBroker
public class WebSocketConfig
  extends AbstractSessionWebSocketMessageBrokerConfigurer<Session> { ❶

 protected void configureStompEndpoints(StompEndpointRegistry registry) { ❷
  registry.addEndpoint("/messages").withSockJS();
 }

 public void configureMessageBroker(MessageBrokerRegistry registry) {
  registry.enableSimpleBroker("/queue/", "/topic/");
  registry.setApplicationDestinationPrefixes("/app");
 }
}
```

To hook in the Spring Session support we only need to change two things:

❶ Instead of extending `AbstractWebSocketMessageBrokerConfigurer` we extend `AbstractSessionWebSocketMessageBrokerConfigurer`

❷ We rename the `registerStompEndpoints` method to `configureStompEndpoints`

What does `AbstractSessionWebSocketMessageBrokerConfigurer` do behind the scenes?

- `WebSocketConnectHandlerDecoratorFactory` is added as a `WebSocketHandlerDecoratorFactory` to `WebSocketTransportRegistration`. This ensures a custom `SessionConnectEvent` is fired that contains the `WebSocketSession`. The `WebSocketSession` is necessary to terminate any WebSocket connections that are still open when a Spring Session is terminated.
- `SessionRepositoryMessageInterceptor` is added as a `HandshakeInterceptor` to every `StompWebSocketEndpointRegistration`. This ensures that the Session is added to the WebSocket properties to enable updating the last accessed time.
- `SessionRepositoryMessageInterceptor` is added as a `ChannelInterceptor` to our inbound `ChannelRegistration`. This ensures that every time an inbound message is received, that the last accessed time of our Spring Session is updated.
- `WebSocketRegistryListener` is created as a Spring Bean. This ensures that we have a mapping of all of the Session id to the corresponding WebSocket connections. By maintaining this mapping, we can close all the WebSocket connections when a Spring Session (HttpSession) is terminated.

6. Spring Security Integration

Spring Session provides integration with Spring Security.

6.1 Spring Security Remember-Me Support

Spring Session provides integration with Spring Security's Remember-Me Authentication. The support will:

- Change the session expiration length
- Ensure the session cookie expires at `Integer.MAX_VALUE`. The cookie expiration is set to the largest possible value because the cookie is only set when the session is created. If it were set to the same value as the session expiration, then the session would get renewed when the user used it but the cookie expiration would not be updated causing the expiration to be fixed.

To configure Spring Session with Spring Security in Java Configuration use the following as a guide:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
 http
  // ... additional configuration ...
  .rememberMe()
   .rememberMeServices(rememberMeServices());
}

@Bean
RememberMeServices rememberMeServices() {
 SpringSessionRememberMeServices rememberMeServices =
   new SpringSessionRememberMeServices();
 // optionally customize
 rememberMeServices.setAlwaysRemember(true);
 return rememberMeServices;
}
```

An XML based configuration would look something like this:

```
<security:http>
 <!-- ... -->
 <security:form-login />
 <security:remember-me services-ref="rememberMeServices"/>
</security:http>

<bean id="rememberMeServices"
 class="org.springframework.session.security.web.authentication.SpringSessionRememberMeServices"
 p:alwaysRemember="true"/>
```

6.2 Spring Security Concurrent Session Control

Spring Session provides integration with Spring Security to support its concurrent session control. This allows limiting the number of active sessions that a single user can have concurrently, but unlike the default Spring Security support this will also work in a clustered environment. This is done by providing a custom implementation of Spring Security's `SessionRegistry` interface.

When using Spring Security's Java config DSL, you can configure the custom `SessionRegistry` through the `SessionManagementConfigurer` like this:

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

 @Autowired
 private FindByIndexNameSessionRepository<Session> sessionRepository;

 @Override
 protected void configure(HttpSecurity http) throws Exception {
  // @formatter:off
  http
   // other config goes here...
   .sessionManagement()
    .maximumSessions(2)
    .sessionRegistry(sessionRegistry());
  // @formatter:on
 }

 @Bean
 SpringSessionBackedSessionRegistry sessionRegistry() {
  return new SpringSessionBackedSessionRegistry<>(this.sessionRepository);
 }
}
```

This assumes that you've also configured Spring Session to provide a `FindByIndexNameSessionRepository` that returns `Session` instances.

When using XML configuration, it would look something like this:

```
<security:http>
 <!-- other config goes here... -->
 <security:session-management>
  <security:concurrency-control max-sessions="2" session-registry-ref="sessionRegistry"/>
 </security:session-management>
</security:http>

<bean id="sessionRegistry"
   class="org.springframework.session.security.SpringSessionBackedSessionRegistry">
 <constructor-arg ref="sessionRepository"/>
</bean>
```

This assumes that your Spring Session `SessionRegistry` bean is called `sessionRegistry`, which is the name used by all `SpringHttpSessionConfiguration` subclasses.

6.3 Limitations

Spring Session's implementation of Spring Security's `SessionRegistry` interface does not support the `getAllPrincipals` method, as this information cannot be retrieved using Spring Session. This method is never called by Spring Security, so this only affects applications that access the `SessionRegistry` themselves.

7. API Documentation

You can browse the complete Javadoc online. The key APIs are described below:

7.1 Session

A `Session` is a simplified `Map` of name value pairs.

Typical usage might look like the following:

```
public class RepositoryDemo<S extends Session> {
 private SessionRepository<S> repository; ❶

 public void demo() {
  S toSave = this.repository.createSession(); ❷

  ❸
  User rwinch = new User("rwinch");
  toSave.setAttribute(ATTR_USER, rwinch);

  this.repository.save(toSave); ❹

  S session = this.repository.findById(toSave.getId()); ❺

  ❻
  Optional<User> user = session.getAttribute(ATTR_USER);
  assertThat(user.orElse(null)).isEqualTo(rwinch);
 }

 // ... setter methods ...
}
```

❶ We create a `SessionRepository` instance with a generic type, S, that extends `Session`. The generic type is defined in our class.

❷ We create a new `Session` using our `SessionRepository` and assign it to a variable of type S.

❸ We interact with the `Session`. In our example, we demonstrate saving a `User` to the `Session`.

❹ We now save the `Session`. This is why we needed the generic type S. The `SessionRepository` only allows saving `Session` instances that were created or retrieved using the same `SessionRepository`. This allows for the `SessionRepository` to make implementation specific optimizations (i.e. only writing attributes that have changed).

❺ We retrieve the `Session` from the `SessionRepository`.

❻ We obtain the persisted `User` from our `Session` without the need for explicitly casting our attribute.

`Session` API also provides attributes related to the `Session` instance's expiration.

Typical usage might look like the following:

```
public class ExpiringRepositoryDemo<S extends Session> {
 private SessionRepository<S> repository; ❶

 public void demo() {
  S toSave = this.repository.createSession(); ❷
  // ...
  toSave.setMaxInactiveInterval(Duration.ofSeconds(30)); ❸

  this.repository.save(toSave); ❹

  S session = this.repository.findById(toSave.getId()); ❺
  // ...
 }

 // ... setter methods ...
}
```

❶ We create a `SessionRepository` instance with a generic type, `S`, that extends `Session`. The generic type is defined in our class.

❷ We create a new `Session` using our `SessionRepository` and assign it to a variable of type `S`.

❸ We interact with the `Session`. In our example, we demonstrate updating the amount of time the `Session` can be inactive before it expires.

❹ We now save the `Session`. This is why we needed the generic type `S`. The `SessionRepository` only allows saving `Session` instances that were created or retrieved using the same `SessionRepository`. This allows for the `SessionRepository` to make implementation specific optimizations (i.e. only writing attributes that have changed). The last accessed time is automatically updated when the `Session` is saved.

❺ We retrieve the `Session` from the `SessionRepository`. If the `Session` were expired, the result would be null.

7.2 SessionRepository

A `SessionRepository` is in charge of creating, retrieving, and persisting `Session` instances.

If possible, developers should not interact directly with a `SessionRepository` or a `Session`. Instead, developers should prefer interacting with `SessionRepository` and `Session` indirectly through the HttpSession and WebSocket integration.

7.3 FindByIndexNameSessionRepository

Spring Session's most basic API for using a `Session` is the `SessionRepository`. This API is intentionally very simple, so that it is easy to provide additional implementations with basic functionality.

Some `SessionRepository` implementations may choose to implement `FindByIndexNameSessionRepository` also. For example, Spring's Redis support implements `FindByIndexNameSessionRepository`.

The `FindByIndexNameSessionRepository` adds a single method to look up all the sessions for a particular user. This is done by ensuring that the session attribute with the name `FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME` is populated with the username. It is the responsibility of the developer to ensure the attribute is populated since Spring Session is not aware of the authentication mechanism being used. An example of how this might be used can be seen below:

```
String username = "username";
this.session.setAttribute(
  FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME, username);
```

> **Note**
>
> Some implementations of `FindByIndexNameSessionRepository` will provide hooks to automatically index other session attributes. For example, many implementations will automatically ensure the current Spring Security user name is indexed with the index name `FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME`.

Once the session is indexed, it can be found using the following:

```
String username = "username";
Map<String, Session> sessionIdToSession = this.sessionRepository
  .findByIndexNameAndIndexValue(
    FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME,
    username);
```

7.4 EnableSpringHttpSession

The `@EnableSpringHttpSession` annotation can be added to an `@Configuration` class to expose the `SessionRepositoryFilter` as a bean named "springSessionRepositoryFilter". In order to leverage the annotation, a single `SessionRepository` bean must be provided. For example:

```
@EnableSpringHttpSession
@Configuration
public class SpringHttpSessionConfig {
 @Bean
 public MapSessionRepository sessionRepository() {
  return new MapSessionRepository();
 }
}
```

It is important to note that no infrastructure for session expirations is configured for you out of the box. This is because things like session expiration are highly implementation dependent. This means if you require cleaning up expired sessions, you are responsible for cleaning up the expired sessions.

7.5 RedisOperationsSessionRepository

`RedisOperationsSessionRepository` is a `SessionRepository` that is implemented using Spring Data's `RedisOperations`. In a web environment, this is typically used in combination with `SessionRepositoryFilter`. The implementation supports `SessionDestroyedEvent` and `SessionCreatedEvent` through `SessionMessageListener`.

Instantiating a RedisOperationsSessionRepository

A typical example of how to create a new instance can be seen below:

```
LettuceConnectionFactory factory = new LettuceConnectionFactory();
SessionRepository<? extends Session> repository = new RedisOperationsSessionRepository(
  factory);
```

For additional information on how to create a `RedisConnectionFactory`, refer to the Spring Data Redis Reference.

EnableRedisHttpSession

In a web environment, the simplest way to create a new `RedisOperationsSessionRepository` is to use `@EnableRedisHttpSession`. Complete example usage can be found in the Chapter 3, *Samples and Guides (Start Here)* You can use the following attributes to customize the configuration:

- **maxInactiveIntervalInSeconds** - the amount of time before the session will expire in seconds
- **redisNamespace** - allows configuring an application specific namespace for the sessions. Redis keys and channel ids will start with the prefix of `spring:session:<redisNamespace>:`.
- **redisFlushMode** - allows specifying when data will be written to Redis. The default is only when `save` is invoked on `SessionRepository`. A value of `RedisFlushMode.IMMEDIATE` will write to Redis as soon as possible.

Custom RedisSerializer

You can customize the serialization by creating a Bean named `springSessionDefaultRedisSerializer` that implements `RedisSerializer<Object>`.

Redis TaskExecutor

`RedisOperationsSessionRepository` is subscribed to receive events from redis using a `RedisMessageListenerContainer`. You can customize the way those events are dispatched, by creating a Bean named `springSessionRedisTaskExecutor` and/or a Bean `springSessionRedisSubscriptionExecutor`. More details on configuring redis task executors can be found here.

Storage Details

The sections below outline how Redis is updated for each operation. An example of creating a new session can be found below. The subsequent sections describe the details.

```
HMSET spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe creationTime 1404360000000 \
 maxInactiveInterval 1800 \
 lastAccessedTime 1404360000000 \
 sessionAttr:attrName someAttrValue \
 sessionAttr2:attrName someAttrValue2
EXPIRE spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe 2100
APPEND spring:session:sessions:expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe ""
EXPIRE spring:session:sessions:expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe 1800
SADD spring:session:expirations:1439245080000 expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe
EXPIRE spring:session:expirations1439245080000 2100
```

Saving a Session

Each session is stored in Redis as a Hash. Each session is set and updated using the HMSET command. An example of how each session is stored can be seen below.

```
HMSET spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe creationTime 1404360000000 \
 maxInactiveInterval 1800 \
 lastAccessedTime 1404360000000 \
 sessionAttr:attrName someAttrValue \
 sessionAttr2:attrName someAttrValue2
```

In this example, the session following statements are true about the session:

- The session id is 33fdd1b6-b496-4b33-9f7d-df96679d32fe
- The session was created at 1404360000000 in milliseconds since midnight of 1/1/1970 GMT.
- The session expires in 1800 seconds (30 minutes).

- The session was last accessed at 1404360000000 in milliseconds since midnight of 1/1/1970 GMT.
- The session has two attributes. The first is "attrName" with the value of "someAttrValue". The second session attribute is named "attrName2" with the value of "someAttrValue2".

Optimized Writes

The `Session` instances managed by `RedisOperationsSessionRepository` keeps track of the properties that have changed and only updates those. This means if an attribute is written once and read many times we only need to write that attribute once. For example, assume the session attribute "sessionAttr2" from earlier was updated. The following would be executed upon saving:

```
HMSET spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe sessionAttr:attrName2 newValue
```

Session Expiration

An expiration is associated to each session using the EXPIRE command based upon the `Session.getMaxInactiveInterval()`. For example:

```
EXPIRE spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe 2100
```

You will note that the expiration that is set is 5 minutes after the session actually expires. This is necessary so that the value of the session can be accessed when the session expires. An expiration is set on the session itself five minutes after it actually expires to ensure it is cleaned up, but only after we perform any necessary processing.

> **Note**
>
> The `SessionRepository.findById(String)` method ensures that no expired sessions will be returned. This means there is no need to check the expiration before using a session.

Spring Session relies on the delete and expired keyspace notifications from Redis to fire a SessionDeletedEvent and SessionExpiredEvent respectively. It is the `SessionDeletedEvent` or `SessionExpiredEvent` that ensures resources associated with the Session are cleaned up. For example, when using Spring Session's WebSocket support the Redis expired or delete event is what triggers any WebSocket connections associated with the session to be closed.

Expiration is not tracked directly on the session key itself since this would mean the session data would no longer be available. Instead a special session expires key is used. In our example the expires key is:

```
APPEND spring:session:sessions:expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe ""
EXPIRE spring:session:sessions:expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe 1800
```

When a session expires key is deleted or expires, the keyspace notification triggers a lookup of the actual session and a SessionDestroyedEvent is fired.

One problem with relying on Redis expiration exclusively is that Redis makes no guarantee of when the expired event will be fired if they key has not been accessed. Specifically the background task that Redis uses to clean up expired keys is a low priority task and may not trigger the key expiration. For additional details see Timing of expired events section in the Redis documentation.

To circumvent the fact that expired events are not guaranteed to happen we can ensure that each key is accessed when it is expected to expire. This means that if the TTL is expired on the key, Redis will remove the key and fire the expired event when we try to access they key.

For this reason, each session expiration is also tracked to the nearest minute. This allows a background task to access the potentially expired sessions to ensure that Redis expired events are fired in a more deterministic fashion. For example:

```
SADD spring:session:expirations:1439245080000 expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe
EXPIRE spring:session:expirations1439245080000 2100
```

The background task will then use these mappings to explicitly request each key. By accessing they key, rather than deleting it, we ensure that Redis deletes the key for us only if the TTL is expired.

> **Note**
>
> We do not explicitly delete the keys since in some instances there may be a race condition that incorrectly identifies a key as expired when it is not. Short of using distributed locks (which would kill our performance) there is no way to ensure the consistency of the expiration mapping. By simply accessing the key, we ensure that the key is only removed if the TTL on that key is expired.

SessionDeletedEvent and SessionExpiredEvent

`SessionDeletedEvent` and `SessionExpiredEvent` are both types of `SessionDestroyedEvent`.

`RedisOperationsSessionRepository` supports firing a `SessionDeletedEvent` whenever a `Session` is deleted or a `SessionExpiredEvent` when it expires. This is necessary to ensure resources associated with the `Session` are properly cleaned up.

For example, when integrating with WebSockets the `SessionDestroyedEvent` is in charge of closing any active WebSocket connections.

Firing `SessionDeletedEvent` or `SessionExpiredEvent` is made available through the `SessionMessageListener` which listens to Redis Keyspace events. In order for this to work, Redis Keyspace events for Generic commands and Expired events needs to be enabled. For example:

```
redis-cli config set notify-keyspace-events Egx
```

If you are using `@EnableRedisHttpSession` the `SessionMessageListener` and enabling the necessary Redis Keyspace events is done automatically. However, in a secured Redis enviornment the config command is disabled. This means that Spring Session cannot configure Redis Keyspace events for you. To disable the automatic configuration add `ConfigureRedisAction.NO_OP` as a bean.

For example, Java Configuration can use the following:

```
@Bean
public static ConfigureRedisAction configureRedisAction() {
 return ConfigureRedisAction.NO_OP;
}
```

XML Configuration can use the following:

```
<util:constant
 static-field="org.springframework.session.data.redis.config.ConfigureRedisAction.NO_OP"/>
```

SessionCreatedEvent

When a session is created an event is sent to Redis with the channel of `spring:session:channel:created:33fdd1b6-b496-4b33-9f7d-df96679d32fe` such that

`33fdd1b6-b496-4b33-9f7d-df96679d32fe` is the session id. The body of the event will be the session that was created.

If registered as a MessageListener (default), then `RedisOperationsSessionRepository` will then translate the Redis message into a `SessionCreatedEvent`.

Viewing the Session in Redis

After installing redis-cli, you can inspect the values in Redis using the redis-cli. For example, enter the following into a terminal:

```
$ redis-cli
redis 127.0.0.1:6379> keys *
1) "spring:session:sessions:4fc39ce3-63b3-4e17-b1c4-5e1ed96fb021" ❶
2) "spring:session:expirations:1418772300000" ❷
```

❶ The suffix of this key is the session identifier of the Spring Session.
❷ This key contains all the session ids that should be deleted at the time `1418772300000`.

You can also view the attributes of each session.

```
redis 127.0.0.1:6379> hkeys spring:session:sessions:4fc39ce3-63b3-4e17-b1c4-5e1ed96fb021
1) "lastAccessedTime"
2) "creationTime"
3) "maxInactiveInterval"
4) "sessionAttr:username"
redis 127.0.0.1:6379> hget spring:session:sessions:4fc39ce3-63b3-4e17-b1c4-5e1ed96fb021
 sessionAttr:username
"\xac\xed\x00\x05t\x00\x03rob"
```

7.6 MapSessionRepository

The `MapSessionRepository` allows for persisting `Session` in a `Map` with the key being the `Session` id and the value being the `Session`. The implementation can be used with a `ConcurrentHashMap` as a testing or convenience mechanism. Alternatively, it can be used with distributed `Map` implementations. For example, it can be used with Hazelcast.

Instantiating MapSessionRepository

Creating a new instance is as simple as:

```
SessionRepository<? extends Session> repository = new MapSessionRepository();
```

Using Spring Session and Hazlecast

The Hazelcast Sample is a complete application demonstrating using Spring Session with Hazelcast.

To run it use the following:

```
./gradlew :samples:hazelcast:tomcatRun
```

The Hazelcast Spring Sample is a complete application demonstrating using Spring Session with Hazelcast and Spring Security.

It includes example Hazelcast `MapListener` implementations that support firing `SessionCreatedEvent`, `SessionDeletedEvent` and `SessionExpiredEvent`.

To run it use the following:

```
./gradlew :samples:hazelcast-spring:tomcatRun
```

7.7 JdbcOperationsSessionRepository

`JdbcOperationsSessionRepository` is a `SessionRepository` implementation that uses Spring's `JdbcOperations` to store sessions in a relational database. In a web environment, this is typically used in combination with `SessionRepositoryFilter`. Please note that this implementation does not support publishing of session events.

Instantiating a JdbcOperationsSessionRepository

A typical example of how to create a new instance can be seen below:

```
JdbcTemplate jdbcTemplate = new JdbcTemplate();

// ... configure JdbcTemplate ...

PlatformTransactionManager transactionManager = new DataSourceTransactionManager();

// ... configure transactionManager ...

SessionRepository<? extends Session> repository =
  new JdbcOperationsSessionRepository(jdbcTemplate, transactionManager);
```

For additional information on how to create and configure `JdbcTemplate` and `PlatformTransactionManager`, refer to the Spring Framework Reference Documentation.

EnableJdbcHttpSession

In a web environment, the simplest way to create a new `JdbcOperationsSessionRepository` is to use `@EnableJdbcHttpSession`. Complete example usage can be found in the Chapter 3, *Samples and Guides (Start Here)* You can use the following attributes to customize the configuration:

- **tableName** - the name of database table used by Spring Session to store sessions
- **maxInactiveIntervalInSeconds** - the amount of time before the session will expire in seconds

Custom LobHandler

You can customize the BLOB handling by creating a Bean named `springSessionLobHandler` that implements `LobHandler`.

Custom ConversionService

You can customize the default serialization and deserialization of the session by providing a `ConversionService` instance. When working in a typical Spring environment, the default `ConversionService` Bean (named `conversionService`) will be automatically picked up and used for serialization and deserialization. However, you can override the default `ConversionService` by providing a Bean named `springSessionConversionService`.

Storage Details

By default, this implementation uses `SPRING_SESSION` and `SPRING_SESSION_ATTRIBUTES` tables to store sessions. Note that the table name can be easily customized as already described. In

that case the table used to store attributes will be named using the provided table name, suffixed with `_ATTRIBUTES`. If further customizations are needed, SQL queries used by the repository can be customized using `set*Query` setter methods. In this case you need to manually configure the `sessionRepository` bean.

Due to the differences between the various database vendors, especially when it comes to storing binary data, make sure to use SQL script specific to your database. Scripts for most major database vendors are packaged as `org/springframework/session/jdbc/schema-*.sql`, where `*` is the target database type.

For example, with PostgreSQL database you would use the following schema script:

```
CREATE TABLE SPRING_SESSION (
 SESSION_ID CHAR(36) NOT NULL,
 CREATION_TIME BIGINT NOT NULL,
 LAST_ACCESS_TIME BIGINT NOT NULL,
 MAX_INACTIVE_INTERVAL INT NOT NULL,
 PRINCIPAL_NAME VARCHAR(100),
 CONSTRAINT SPRING_SESSION_PK PRIMARY KEY (SESSION_ID)
);

CREATE INDEX SPRING_SESSION_IX1 ON SPRING_SESSION (LAST_ACCESS_TIME);

CREATE TABLE SPRING_SESSION_ATTRIBUTES (
 SESSION_ID CHAR(36) NOT NULL,
 ATTRIBUTE_NAME VARCHAR(200) NOT NULL,
 ATTRIBUTE_BYTES BYTEA NOT NULL,
 CONSTRAINT SPRING_SESSION_ATTRIBUTES_PK PRIMARY KEY (SESSION_ID, ATTRIBUTE_NAME),
 CONSTRAINT SPRING_SESSION_ATTRIBUTES_FK FOREIGN KEY (SESSION_ID) REFERENCES
 SPRING_SESSION(SESSION_ID) ON DELETE CASCADE
);

CREATE INDEX SPRING_SESSION_ATTRIBUTES_IX1 ON SPRING_SESSION_ATTRIBUTES (SESSION_ID);
```

And with MySQL database:

```
CREATE TABLE SPRING_SESSION (
 SESSION_ID CHAR(36) NOT NULL,
 CREATION_TIME BIGINT NOT NULL,
 LAST_ACCESS_TIME BIGINT NOT NULL,
 MAX_INACTIVE_INTERVAL INT NOT NULL,
 PRINCIPAL_NAME VARCHAR(100),
 CONSTRAINT SPRING_SESSION_PK PRIMARY KEY (SESSION_ID)
) ENGINE=InnoDB;

CREATE INDEX SPRING_SESSION_IX1 ON SPRING_SESSION (LAST_ACCESS_TIME);

CREATE TABLE SPRING_SESSION_ATTRIBUTES (
 SESSION_ID CHAR(36) NOT NULL,
 ATTRIBUTE_NAME VARCHAR(200) NOT NULL,
 ATTRIBUTE_BYTES BLOB NOT NULL,
 CONSTRAINT SPRING_SESSION_ATTRIBUTES_PK PRIMARY KEY (SESSION_ID, ATTRIBUTE_NAME),
 CONSTRAINT SPRING_SESSION_ATTRIBUTES_FK FOREIGN KEY (SESSION_ID) REFERENCES
 SPRING_SESSION(SESSION_ID) ON DELETE CASCADE
) ENGINE=InnoDB;

CREATE INDEX SPRING_SESSION_ATTRIBUTES_IX1 ON SPRING_SESSION_ATTRIBUTES (SESSION_ID);
```

Transaction management

All JDBC operations in `JdbcOperationsSessionRepository` are executed in a transactional manner. Transactions are executed with propagation set to `REQUIRES_NEW` in order to avoid unexpected behavior due to interference with existing transactions (for example, executing `save` operation in a thread that already participates in a read-only transaction).

7.8 HazelcastSessionRepository

`HazelcastSessionRepository` is a `SessionRepository` implementation that stores sessions in Hazelcast's distributed `IMap`. In a web environment, this is typically used in combination with `SessionRepositoryFilter`.

Instantiating a HazelcastSessionRepository

A typical example of how to create a new instance can be seen below:

```
Config config = new Config();

// ... configure Hazelcast ...

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance(config);

IMap<String, MapSession> sessions = hazelcastInstance
  .getMap("spring:session:sessions");

HazelcastSessionRepository repository =
  new HazelcastSessionRepository(sessions);
```

For additional information on how to create and configure Hazelcast instance, refer to the Hazelcast documentation.

EnableHazelcastHttpSession

If you wish to use Hazelcast as your backing source for the `SessionRepository`, then the `@EnableHazelcastHttpSession` annotation can be added to an `@Configuration` class. This extends the functionality provided by the `@EnableSpringHttpSession` annotation but makes the `SessionRepository` for you in Hazelcast. You must provide a single `HazelcastInstance` bean for the configuration to work. Complete configuration example can be found in the Chapter 3, *Samples and Guides (Start Here)*

Basic Customization

You can use the following attributes on `@EnableHazelcastHttpSession` to customize the configuration:

- **maxInactiveIntervalInSeconds** - the amount of time before the session will expire in seconds. Default is 1800 seconds (30 minutes)
- **sessionMapName** - the name of the distributed `Map` that will be used in Hazelcast to store the session data.

Session Events

Using a `MapListener` to respond to entries being added, evicted, and removed from the distributed `Map`, these events will trigger publishing `SessionCreatedEvent`, `SessionExpiredEvent`, and `SessionDeletedEvent` events respectively using the `ApplicationEventPublisher`.

Storage Details

Sessions will be stored in a distributed `IMap` in Hazelcast. The `IMap` interface methods will be used to `get()` and `put()` Sessions. Additionally, `values()` method is used to support `FindByIndexNameSessionRepository#findByIndexNameAndIndexValue` operation, together

with appropriate `ValueExtractor` that needs to be registered with Hazelcast. Refer to Hazelcast Spring Sample for more details on this configuration. The expiration of a session in the `IMap` is handled by Hazelcast's support for setting the time to live on an entry when it is `put()` into the `IMap`. Entries (sessions) that have been idle longer than the time to live will be automatically removed from the `IMap`.

You shouldn't need to configure any settings such as `max-idle-seconds` or `time-to-live-seconds` for the `IMap` within the Hazelcast configuration.

Note that if you use Hazelcast's `MapStore` to persist your sessions `IMap` there are some limitations when reloading the sessions from `MapStore`:

- reload triggers `EntryAddedListener` which results in `SessionCreatedEvent` being re-published
- reload uses default TTL for a given `IMap` which results in sessions losing their original TTL

8. Spring Session Community

We are glad to consider you a part of our community. Please find additional information below.

8.1 Support

You can get help by asking questions on StackOverflow with the tag spring-session. Similarly we encourage helping others by answering questions on StackOverflow.

8.2 Source Code

Our source code can be found on github at https://github.com/spring-projects/spring-session/

8.3 Issue Tracking

We track issues in github issues at https://github.com/spring-projects/spring-session/issues

8.4 Contributing

We appreciate Pull Requests.

8.5 License

Spring Session is Open Source software released under the Apache 2.0 license.

8.6 Community Extensions

Name	Location
Spring Session OrientDB	https://github.com/maseev/spring-session-orientdb
Spring Session Infinispan	http://infinispan.org/docs/dev/user_guide/user_guide.html#externalizing_session_using_spring_session

9. Minimum Requirements

The minimum requirements for Spring Session are:

- Java 8+
- If you are running in a Servlet Container (not required), Servlet 3.1+
- If you are using other Spring libraries (not required), the minimum required version is Spring 5.0.x.
- `@EnableRedisHttpSession` requires Redis 2.8+. This is necessary to support Session Expiration
- `@EnableHazelcastHttpSession` requires Hazelcast 3.6+. This is necessary to support `FindByIndexNameSessionRepository`

> **Note**
>
> At its core Spring Session only has a required dependency on `spring-jcl`. For an example of using Spring Session without any other Spring dependencies, refer to the hazelcast sample application.