Spring Session

John Blum

Version 2.5.4

Table of Contents

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

This documentation is also available as HTML.

Chapter 1. Introduction

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

- **HttpSession** - enables the `HttpSession` to be clustered (i.e. replicated for highly availability) without being tied to an application container specific solution.
- **REST API** - allows the session ID to be provided in the protocol header to work with RESTful APIs.
- **WebSocket** - provides the ability to keep the `HttpSession` alive when receiving WebSocket messages.
- **WebSession** - allows replacing the Spring WebFlux's `WebSession` in an application container neutral way.

In a nutshell, Spring Session replaces the `javax.servlet.http.HttpSession` in an application container neutral way by supplying a more common and robust session implementation backing the `HttpSession`.

Chapter 2. Samples and Guides (Start Here)

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

Table 1. Sample Application using Spring Boot

Source	Description	Guide
HttpSession with Spring Boot and Apache Geode	Demonstrates how to use Spring Session to manage the `HttpSession` with Apache Geode in a Spring Boot application using a Client/Server topology.	HttpSession with Spring Boot and Apache Geode Guide
HttpSession with Spring Boot and Apache Geode using Gfsh	Demonstrates how to use Spring Session to manage the `HttpSession` with Apache Geode in a Spring Boot application using a Client/Server topology. Additionally configures and uses Apache Geode's *DataSerialization* framework.	HttpSession with Spring Boot and Apache Geode using Gfsh Guide
HttpSession with Spring Boot and Apache Geode using Scoped Proxies	Demonstrates how to use Spring Session to manage the `HttpSession` with Apache Geode in a Spring Boot application using a Client/Server topology. The application also makes use of Spring Request and Session Scoped Proxy beans.	HttpSession with Spring Boot and Apache Geode using Scoped Proxies Guide

Table 2. Sample Applications using Spring's Java-based configuration

Source	Description	Guide
HttpSession with Apache Geode (Client/Server)	Demonstrates how to use Spring Session to manage the `HttpSession` with Apache Geode using a Client/Server topology.	HttpSession with Apache Geode (Client/Server) Guide
HttpSession with Apache Geode (P2P)	Demonstrates how to use Spring Session to manage the `HttpSession` with Apache Geode using a P2P topology.	HttpSession with Apache Geode (P2P) Guide

Table 3. Sample Applications using Spring's XML-based configuration

Source	Description	Guide
HttpSession with Apache Geode (Client/Server)	Demonstrates how to use Spring Session to manage the `HttpSession` with Apache Geode using a Client/Server topology.	HttpSession with Apache Geode (Client/Server) Guide
HttpSession with Apache Geode (P2P)	Demonstrates how to use Spring Session to manage the `HttpSession` with Apache Geode using a P2P topology.	HttpSession with Apache Geode (P2P) Guide

Chapter 3. HttpSession Integration

Spring Session provides transparent integration with `javax.servlet.http.HttpSession`. This means that developers can replace the `HttpSession` implementation with an implementation that is backed by Spring Session.

Spring Session enables multiple different data store providers (e.g. like Apache Geode) to be plugged in in order to manage the `HttpSession` state.

3.1. Why Spring Session & HttpSession?

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

- **HttpSession** - enables the `HttpSession` to be clustered (i.e. replicated for highly availability) without being tied to an application container specific solution.
- **REST API** - allows the session ID to be provided in the protocol header to work with RESTful APIs.
- **WebSocket** - provides the ability to keep the `HttpSession` alive when receiving WebSocket messages.
- **WebSession** - allows replacing the Spring WebFlux's `WebSession` in an application container neutral way.

3.2. HttpSession Management with Apache Geode

When Apache Geode is used with Spring Session, a web application's `javax.servlet.http.HttpSession` can be replaced with a **clustered** implementation managed by Apache Geode and conveniently accessed using Spring Session's API.

The two most common topologies for managing session state using Apache Geode include:

- Client-Server
- Peer-To-Peer (P2P)

Additionally, Apache Geode supports site-to-site replication using the WAN topology. The ability to configure and use Apache Geode's WAN functionality is independent of Spring Session, and beyond the scope of this document.

More details on configuring Apache Geode WAN functionality using Spring Data for Apache Geode can be found here.

3.2.1. Apache Geode Client-Server

The Client-Server topology will likely be the most common configuration choice among users when using Apache Geode as a provider in Spring Session since a Apache Geode server has significantly different and unique JVM heap requirements than compared to the application. Using a Client-

Server topology enables an application to manage (e.g. replicate) application state independently from other application processes.

In a Client-Server topology, an application using Spring Session will open 1 or more connections to a remote cluster of Apache Geode servers that will manage access to all `HttpSession` state.

You can configure a Client-Server topology with either:

- Java-based Configuration
- XML-based Configuration

Apache Geode Client-Server Java-based Configuration

This section describes how to configure Apache Geode's Client-Server topology to manage `HttpSession` state with Java-based configuration.

The HttpSession with Apache Geode (Client-Server) provides a working sample demonstrating how to integrate Spring Session with Apache Geode to manage `HttpSession` state using Java configuration. You can read through the basic steps of integration below, but you are encouraged to follow along in the detailed *`HttpSession` with Apache Geode (Client-Server) Guide* when integrating with your own application.

Spring Java Configuration

After adding the required dependencies and repository declarations, we can create the Spring configuration. The Spring configuration is responsible for creating a Servlet `Filter` that replaces the `HttpSession` with an implementation backed by Spring Session and Apache Geode.

Client Configuration

Add the following Spring configuration:

```
@ClientCacheApplication(name = "SpringSessionDataGeodeJavaConfigSampleClient",
logLevel = "error",
    readTimeout = 15000, retryAttempts = 1, subscriptionEnabled = true) ①
@EnableGemFireHttpSession(poolName = "DEFAULT") ②
public class ClientConfig extends ClientServerIntegrationTestsSupport {

    @Bean
    ClientCacheConfigurer gemfireServerReadyConfigurer( ③
            @Value("${spring.data.gemfire.cache.server.port:40404}") int
cacheServerPort) {

        return (beanName, clientCacheFactoryBean) -> waitForServerToStart("localhost",
cacheServerPort);
    }
}
```

① First, we declare our Web application to be an Apache Geode cache client by annotating our `ClientConfig` class with `@ClientCacheApplication`. Additionally, we adjust a few basic, "DEFAULT" `Pool` settings (e.g. `readTimeout`).

② `@EnableGemFireHttpSession` creates a Spring bean named `springSessionRepositoryFilter` that implements `javax.servlet.Filter`. The filter replaces the `HttpSession` with an implementation provided by Spring Session and backed by Apache Geode. Additionally, the configuration will also create the necessary client-side `Region` (by default, "ClusteredSpringSessions`, which is a `PROXY Region`) corresponding to the same server-side `Region` by name. All session state is sent from the client to the server through `Region` data access operations. The client-side `Region` use the "DEFAULT" `Pool`.

③ Then, we wait to ensure the Apache Geode Server is up and running before we proceed. This is only really useful for automated (integration) testing purposes.

In typical Apache Geode production deployments, where the cluster includes potentially hundreds or thousands of servers (a.k.a. data nodes), it is more common for clients to connect to 1 or more Apache Geode Locators running in the same cluster. A Locator passes meta-data to clients about the servers available in the cluster, the individual server load and which servers have the client's data of interest, which is particularly important for direct, single-hop data access and latency-sensitive applications. See more details about the Client/Server Deployment in the Apache Geode User Guide.

For more information on configuring Spring Data Geode, refer to the Reference Guide.

The `@EnableGemFireHttpSession` annotation enables developers to configure certain aspects of both Spring Session and Apache Geode out-of-the-box using the following attributes:

- `clientRegionShortcut` - specifies Apache Geode data management policy on the client with the ClientRegionShortcut (default is `PROXY`). This attribute is only used when configuring the client `Region`.
- `indexableSessionAttributes` - Identifies the Session attributes by name that should be indexed for querying purposes. Only Session attributes explicitly identified by name will be indexed.
- `maxInactiveIntervalInSeconds` - controls *HttpSession* idle-timeout expiration (defaults to **30 minutes**).
- `poolName` - name of the dedicated Apache Geode `Pool` used to connect a client to the cluster of servers. This attribute is only used when the application is a cache client. Defaults to `gemfirePool`.
- `regionName` - specifies the name of the Apache Geode `Region` used to store and manage `HttpSession` state (default is "**ClusteredSpringSessions**").
- `serverRegionShortcut` - specifies Apache Geode data management policy on the server with the RegionShortcut (default is `PARTITION`). This attribute is only used when configuring server `Regions`, or when a P2P topology is employed.

It is important to remember that the Apache Geode client `Region` name must match a server `Region` by the same name if the client `Region` is a `PROXY` or `CACHING_PROXY`. Client and server `Region` names are not required to match if the client `Region` used to store session state is `LOCAL`. However, keep in mind that Session state will not be propagated to the server and you lose all the benefits of using Apache Geode to store and manage distributed, replicated session state information on the servers in a distributed, replicated manner.

Server Configuration

So far, we only covered one side of the equation. We also need an Apache Geode Server for our cache client to talk to and send session state to the server to manage.

In this sample, we will use the following Java configuration to configure and run an Apache Geode Server:

```
@CacheServerApplication(name = "SpringSessionDataGeodeJavaConfigSampleServer",
logLevel = "error") ①
@EnableGemFireHttpSession(maxInactiveIntervalInSeconds = 30) ②
public class GemFireServer {

    @SuppressWarnings("resource")
    public static void main(String[] args) {
        new
AnnotationConfigApplicationContext(GemFireServer.class).registerShutdownHook();
    }
}
```

① First, we use the `@CacheServerApplication` annotation to simplify the creation of a peer cache instance containing with a `CacheServer` for cache clients to connect.

② (Optional) Then, the `GemFireServer` class is annotated with `@EnableGemFireHttpSession` to create the necessary server-side `Region` (by default, "*ClusteredSpringSessions*") used to store `HttpSession` state. This step is optional since the Session `Region` could be created manually, perhaps using external means. Using `@EnableGemFireHttpSession` is convenient and quick.

Apache Geode Client-Server XML-based Configuration

This section describes how to configure Apache Geode's Client-Server topology to manage `HttpSession` state with XML-based configuration.

The HttpSession with Apache Geode (Client-Server) using XML provides a working sample demonstrating how to integrate Spring Session with Apache Geode to manage `HttpSession` state using XML configuration. You can read through the basic steps of integration below, but you are encouraged to follow along in the detailed *`HttpSession` with Apache Geode (Client-Server) using XML Guide* when integrating with your own application.

Spring XML Configuration

After adding the required dependencies and repository declarations, we can create the Spring configuration. The Spring configuration is responsible for creating a `Servlet Filter` that replaces the `javax.servlet.http.HttpSession` with an implementation backed by Spring Session and Apache Geode.

Client Configuration

Add the following Spring configuration:

```
    <context:annotation-config/>

    <context:property-placeholder/>

    <bean class="sample.client.ClientServerReadyBeanPostProcessor"/>

    ①
    <util:properties id="gemfireProperties">
        <prop key="log-level">${spring.data.gemfire.cache.log-level:error}</prop>
    </util:properties>

    ②
    <gfe:client-cache properties-ref="gemfireProperties" pool-name="gemfirePool"/>

    ③
    <gfe:pool read-timeout="15000" retry-attempts="1" subscription-enabled="true">
        <gfe:server host="localhost"
port="${spring.data.gemfire.cache.server.port:40404}"/>
    </gfe:pool>

    ④
    <bean
class="org.springframework.session.data.gemfire.config.annotation.web.http.GemFireHttp
SessionConfiguration"
          p:poolName="DEFAULT"/>
```

① (Optional) First, we can include a `Properties` bean to configure certain aspects of the Apache Geode `ClientCache` using VMware Tanzu GemFire Properties. In this case, we are just setting Apache Geode's "log-level" using an application-specific System property, defaulting to "warning" if unspecified.

② We must create an instance of an Apache Geode `ClientCache`. We initialize it with our `gemfireProperties`.

③ Then we configure a `Pool` of connections to talk to the Apache Geode Server in our Client/Server topology. In our configuration, we use sensible settings for timeouts, number of connections and so on. Also, our `Pool` has been configured to connect directly to the server (using the nested `gfe:server` element).

④ Finally, a `GemFireHttpSessionConfiguration` bean is registered to enable Spring Session

functionality.

In typical Apache Geode production deployments, where the cluster includes potentially hundreds or thousands of servers (a.k.a. data nodes), it is more common for clients to connect to 1 or more Apache Geode Locators running in the same cluster. A Locator passes meta-data to clients about the servers available in the cluster, the individual server load and which servers have the client's data of interest, which is particularly important for direct, single-hop data access and latency-sensitive applications. See more details about the Client/Server Deployment in the Apache Geode User Guide.

For more information on configuring Spring Data for Apache Geode, refer to the Reference Guide.

Server Configuration

So far, we only covered one side of the equation. We also need an Apache Geode Server for our cache client to talk to and send session state to the server to manage.

In this sample, we will use the following XML configuration to spin up an Apache Geode Server:

```
    <context:annotation-config/>

    <context:property-placeholder/>

    ①
    <util:properties id="gemfireProperties">
        <prop key="name">SpringSessionDataGeodeSampleXmlServer</prop>
        <prop key="log-level">${spring.data.gemfire.cache.log-level:error}</prop>
    </util:properties>

    ②
    <gfe:cache properties-ref="gemfireProperties"/>

    ③
    <gfe:cache-server port="${spring.data.gemfire.cache.server.port:40404}"/>

    ④
    <bean
class="org.springframework.session.data.gemfire.config.annotation.web.http.GemFireHttp
SessionConfiguration"
          p:maxInactiveIntervalInSeconds="30"/>
```

① (Optional) First, we can include a `Properties` bean to configure certain aspects of the Apache Geode peer `Cache` using VMware Tanzu GemFire Properties. In this case, we are just setting Apache Geode's "log-level" using an application-specific System property, defaulting to "warning" if unspecified.

② We must configure an Apache Geode peer `Cache` instance. We initialize it with the Apache Geode

properties.

③ Next, we define a `CacheServer` with sensible configuration for `bind-address` and `port` used by our cache client application to connect to the server and send session state.

④ Finally, we enable the same Spring Session functionality we declared in the client XML configuration by registering an instance of `GemFireHttpSessionConfiguration`, except we set the session expiration timeout to **30 seconds**. We explain what this means later.

The Apache Geode Server gets bootstrapped with the following:

```
@Configuration ①
@ImportResource("META-INF/spring/session-server.xml") ②
public class GemFireServer {

    public static void main(String[] args) {
        new
AnnotationConfigApplicationContext(GemFireServer.class).registerShutdownHook();
    }
}
```

Rather than defining a simple Java class with a `main` method, you might consider using Spring Boot instead.

① The `@Configuration` annotation designates this Java class as a source of Spring configuration meta-data using 7.9. Annotation-based container configuration[Spring's annotation configuration support].

② Primarily, the configuration comes from the `META-INF/spring/session-server.xml` file.

XML Servlet Container Initialization

Our Spring XML Configuration created a Spring bean named `springSessionRepositoryFilter` that implements `javax.servlet.Filter` interface. The `springSessionRepositoryFilter` bean is responsible for replacing the `javax.servlet.http.HttpSession` with a custom implementation that is provided by Spring Session and Apache Geode.

In order for our `Filter` to do its magic, we need to instruct Spring to load our `session-client.xml` configuration file.

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/session-client.xml</param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>
```

The ContextLoaderListener reads the `contextConfigLocation` context parameter value and picks up our *session-client.xml* configuration file.

Finally, 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 HTTP request, the `DelegatingFilterProxy` is invoked, which delegates to the `springSessionRepositoryFilter`.

3.2.2. Apache Geode Peer-To-Peer (P2P)

A less common approach is to configure your Spring Session application as a peer member in the Apache Geode cluster using the Peer-To-Peer (P2P) topology. In this configuration, the Spring Session application would be an actual server (or data node) in the Apache Geode cluster, and **not** just a cache client as before.

One advantage to this approach is the proximity of the application to the application's state (i.e. its data), and in particular the `HttpSession` state. However, there are other effective means of accomplishing similar data dependent computations, such as using Apache Geode's Function Execution. Any of Apache Geode's other features can be used when Apache Geode is serving as a provider in Spring Session.

The P2P topology is very useful for testing purposes and for smaller, more focused and self-contained applications, such as those found in a microservices architecture, and will most certainly improve on your application's perceived latency and throughput needs.

You can configure a Peer-To-Peer (P2P) topology with either:

- Java-based Configuration
- XML-based Configuration

Apache Geode Peer-To-Peer (P2P) Java-based Configuration

This section describes how to configure Apache Geode's Peer-To-Peer (P2P) topology to manage `HttpSession` state using Java-based configuration.

The HttpSession with Apache Geode (P2P) provides a working sample demonstrating how to integrate Spring Session with Apache Geode to manage `HttpSession` state using Java configuration. You can read through the basic steps of integration below, but you are encouraged to follow along in the detailed *`HttpSession` with Apache Geode (P2P) Guide* when integrating with your own application.

Spring Java Configuration

After adding the required dependencies and repository declarations, we can create the Spring configuration.

The Spring configuration is responsible for creating a `Servlet Filter` that replaces the `javax.servlet.http.HttpSession` with an implementation backed by Spring Session and Apache Geode.

Add the following Spring configuration:

```
@PeerCacheApplication(name = "SpringSessionDataGeodeJavaConfigP2pSample", logLevel =
"error") ①
@EnableGemFireHttpSession(maxInactiveIntervalInSeconds = 30) ②
public class Config {

}
```

① First, we use the `@PeerCacheApplication` annotation to simplify the creation of a peer cache instance.

② Then, the `Config` class is annotated with `@EnableGemFireHttpSession` to create the necessary server-side `Region` (by default, "*ClusteredSpringSessions*") used to store `HttpSession` state.

For more information on configuring Spring Data for Apache Geode, refer to the Reference Guide.

The `@EnableGemFireHttpSession` annotation enables developers to configure certain aspects of both

Spring Session and Apache Geode out-of-the-box using the following attributes:

- `clientRegionShortcut` - specifies Apache Geode data management policy on the client with the ClientRegionShortcut (default is `PROXY`). This attribute is only used when configuring the client `Region`.
- `indexableSessionAttributes` - Identifies the Session attributes by name that should be indexed for querying purposes. Only Session attributes explicitly identified by name will be indexed.
- `maxInactiveIntervalInSeconds` - controls *HttpSession* idle-timeout expiration (defaults to **30 minutes**).
- `poolName` - name of the dedicated Apache Geode `Pool` used to connect a client to the cluster of servers. This attribute is only used when the application is a cache client. Defaults to `gemfirePool`.
- `regionName` - specifies the name of the Apache Geode `Region` used to store and manage `HttpSession` state (default is "**ClusteredSpringSessions**").
- `serverRegionShortcut` - specifies Apache Geode data management policy on the server with the RegionShortcut (default is `PARTITION`). This attribute is only used when configuring server `Regions`, or when a P2P topology is employed.

Java Servlet Container Initialization

Our <<[httpsession-spring-java-configuration-p2p,Spring Java Configuration>> created a Spring bean named `springSessionRepositoryFilter` that implements `javax.servlet.Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `javax.servlet.http.HttpSession` with a custom implementation backed by Spring Session and Apache Geode.

In order for our `Filter` to do its magic, Spring needs to load our `Config` class. We also need to ensure our Servlet container (i.e. Tomcat) uses our `springSessionRepositoryFilter` on every HTTP request.

Fortunately, Spring Session provides a utility class named `AbstractHttpSessionApplicationInitializer` to make both steps extremely easy.

You can find an example below:

src/main/java/sample/Initializer.java

```
public class Initializer extends AbstractHttpSessionApplicationInitializer { ①

    public Initializer() {
        super(Config.class); ②
    }
}
```

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

① The first step is to extend `AbstractHttpSessionApplicationInitializer`. This ensures that a Spring bean named `springSessionRepositoryFilter` is registered with our Servlet container and used on

every HTTP request.

② `AbstractHttpSessionApplicationInitializer` also provides a mechanism to easily allow Spring to load our `Config` class.

Apache Geode Peer-To-Peer (P2P) XML-based Configuration

This section describes how to configure Apache Geode's Peer-To-Peer (P2P) topology to manage `HttpSession` state using XML-based configuration.

The HttpSession with Apache Geode (P2P) using XML provides a working sample demonstrating how to integrate Spring Session with Apache Geode to manage `HttpSession` state using XML configuration. You can read through the basic steps of integration below, but you are encouraged to follow along in the detailed *`HttpSession` with Apache Geode (P2P) using XML Guide* when integrating with your own application.

Spring XML Configuration

After adding the required dependencies and repository declarations, we can create the Spring configuration.

The Spring configuration is responsible for creating a `Servlet Filter` that replaces the `javax.servlet.http.HttpSession` with an implementation backed by Spring Session and Apache Geode.

Add the following Spring configuration:

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

```
<context:annotation-config/>

<context:property-placeholder/>

①
<util:properties id="gemfireProperties">
    <prop key="name">SpringSessionDataGeodeXmlP2pSample</prop>
    <prop key="log-level">${spring.data.gemfire.cache.log-level:error}</prop>
</util:properties>

②
<gfe:cache properties-ref="gemfireProperties"/>

③
<bean
class="org.springframework.session.data.gemfire.config.annotation.web.http.GemFireHttp
SessionConfiguration"
    p:maxInactiveIntervalInSeconds="30"/>
```

① (Optional) First, we can include a `Properties` bean to configure certain aspects of the Apache Geode peer `Cache` using VMware Tanzu GemFire Properties. In this case, we are just setting

Apache Geode's "log-level" using an application-specific System property, defaulting to "warning" if unspecified.

② We must configure an Apache Geode peer `Cache` instance. We initialize it with the Apache Geode properties.

③ Finally, we enable Spring Session functionality by registering an instance of `GemFireHttpSessionConfiguration`.

For more information on configuring Spring Data for Apache Geode, refer to the Reference Guide.

XML Servlet Container Initialization

The Spring XML Configuration created a Spring bean named `springSessionRepositoryFilter` that implements `javax.servlet.Filter`. The `springSessionRepositoryFilter` bean is responsible for replacing the `javax.servlet.http.HttpSession` with a custom implementation that is backed by Spring Session and Apache Geode.

In order for our `Filter` to do its magic, we need to instruct Spring to load our `session.xml` configuration file.

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` context parameter value and picks up our *session.xml* configuration file.

Finally, we need to ensure that our Servlet container (i.e. Tomcat) uses our `springSessionRepositoryFilter` for every HTTP 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 HTTP request the `DelegatingFilterProxy` is invoked, delegating to the `springSessionRepositoryFilter`.

3.3. Configuring `HttpSession` Management using Apache Geode with Properties

While the `@EnableGemFireHttpSession` annotation is easy to use and convenient when getting started with Spring Session and Apache Geode in your Spring Boot applications, you quickly run into limitations when migrating from one environment to another, for example, like when moving from DEV to QA to PROD.

With the `@EnableGemFireHttpSession` annotation attributes, it is not possible to vary the configuration from one environment to another. Therefore, Spring Session for Apache Geode introduces well-known, documented properties for all the `@EnableGemFireHttpSession` annotation attributes.

Table 4. Well-known, documented properties for the `@EnableGemFireHttpSession` annotation attributes.

Property	Annotation attribute	Description	Default
spring.session.data.gemfire.cache.client.pool.name	`EnableGemFireHttpSession.poolName`	Name of the dedicated Pool used by the client Region storing/accessing Session state.	gemfirePool
spring.session.data.gemfire.cache.client.region.shortcut	`EnableGemFireHttpSession.clientRegionShortcut`	Sets the client Region data management policy in the client-server topology.	ClientRegionShortcut.PROXY
spring.session.data.gemfire.cache.server.region.shortcut	`EnableGemFireHttpSession.serverRegionShortcut`	Sets the peer Region data management policy in the peer-to-peer (P2P) topology.	RegionShortcut.PARTITION

Property	Annotation attribute	Description	Default
spring.session.data.gemfire.session.attributes.indexable	`EnableGemFireHttpSession.indexableSessionAttributes`	Comma-delimited list of Session attributes to indexed in the Session Region.	
spring.session.data.gemfire.session.expiration.bean-name	`EnableGemFireHttpSession.sessionExpirationPolicyBeanName`	Name of the bean in the Spring container implementing the expiration strategy	
spring.session.data.gemfire.session.expiration.max-inactive-interval-seconds	`EnableGemFireHttpSession.maxInactiveIntervalInSeconds`	Session expiration timeout in seconds	1800
spring.session.data.gemfire.session.region.name	`EnableGemFireHttpSession.regionName`	Name of the client or peer Region used to store and access Session state.	ClusteredSpringSessions
spring.session.data.gemfire.session.serializer.bean-name	`EnableGemFireHttpSession.sessionSerializerBeanName`	Name of the bean in the Spring container implementing the serialization strategy	SessionPdxSerializer

All the properties are documented in the `@EnableGemFireHttpSession` annotation attribute Javadoc as well.

Therefore, it is very simple to adjust the configuration of Spring Session when using Apache Geode as your provider by using properties, as follows:

```
@SpringBootApplication
@ClientCacheApplication
@EnableGemFireHttpSession(maxInactiveIntervalInSeconds = 900)
class MySpringSessionApplication {
  // ...
}
```

And then, in `application.properties`:

```
#application.properties
spring.session.data.gemfire.cache.client.region.shortcut=CACHING_PROXY
spring.session.data.gemfire.session.expiration.max-inactive-internval-seconds=3600
```

Any properties explicitly defined override the corresponding `@EnableGemFireHttpSession` annotation attribute.

In the example above, even though the `EnableGemFireHttpSession` annotation `maxInactiveIntervalInSeconds` attribute was set to `900` seconds, or 15 minutes, the corresponding

attribute property (i.e. `spring.session.data.gemfire.session.expiration.max-inactive-interval-seconds`) overrides the value and sets the expiration to `3600` seconds, or 60 minutes.

Keep in mind, properties override the annotation attribute values at runtime.

3.3.1. Properties of Properties

You can even get more sophisticated and configure your properties with other properties, as follows:

```
#application.properties
spring.session.data.gemfire.session.expiration.max-inactive-internval-
seconds=${app.geode.region.expiration.timeout:3600}
```

Additionally, you could use Spring profiles to vary the expiration timeout (or other properties) based on environment or your application, or whatever criteria your application requirements dictate.

Property placeholders and nesting is a feature of the core Spring Framework and not specific to Spring Session or Spring Session for Apache Geode.

3.4. Configuring `HttpSession` Management using Apache Geode with a Configurer

In addition to properties, Spring Session for Apache Geode also allows you to adjust the configuration of Spring Session with Apache Geode using the `SpringSessionGemFireConfigurer` interface. The interface defines a contract containing default methods for each `@EnableGemFireHttpSession` annotation attribute that can be overridden to adjust the configuration.

The `SpringSessionGemFireConfigurer` is similar in concept to Spring Web MVC's Configurer interfaces (e.g. `o.s.web.servlet.config.annotation.WebMvcConfigurer`), which adjusts various aspects of your Web application's configuration on startup, such as configuring async support. The advantage of declaring and implementing a `Configurer` is that it gives you programmatical control over your configuration. This is useful in situations where you need to easily express complex, conditional logic that determines whether the configuration should be applied or not.

For example, to adjust the client Region data management policy and Session expiration timeout as we did previously, use the following:

```
@Configuration
class MySpringSessionConfiguration {

  @Bean
  SpringSessionGemFireConfigurer exampleSpringSessionGemFireConfigurer() {

    return new SpringSessionGemFireConfigurer() {

      @Override
      public ClientRegionShortcut getClientRegionShortcut() {
        return ClientRegionShortcut.CACHING_PROXY;
      }

      @Override
      public int getMaxInactiveIntervalInSeconds() {
        return 3600;
      }
    };
  }
}
```

Of course, this example is not very creative. You could most certainly use more complex logic to determine the configuration of each configuration attribute.

You can be as sophisticated as you like, such as by implementing your `Configurer` in terms of other properties using Spring's `@Value` annotation, as follows:

```
@Configuration
class MySpringSessionConfiguration {

  @Bean
  @Primary
  @Profile("production")
  SpringSessionGemFireConfigurer exampleSpringSessionGemFireConfigurer(
      @Value("${app.geode.region.data-management-policy:CACHING_PROXY}")
ClientRegionShortcut shortcut,
      @Value("${app.geode.region.expiration.timeout:3600}") int timeout) {

    return new SpringSessionGemFireConfigurer() {

      @Override
      public ClientRegionShortcut getClientRegionShortcut() {
        return shortcut;
      }

      @Override
      public int getMaxInactiveIntervalInSeconds() {
        return timeout;
      }
    };
  }
}
```

Spring Boot will resolve `@Value` annotation property placeholder values or SpEL Expressions automatically. However, if you are not using Spring Boot, then you must explicitly register a static `PropertySourcesPlaceholderConfigurer` bean definition.

However, you can only declare 1 `SpringSessionGemFireConfigurer` bean in the Spring container at a time, unless you are also using Spring profiles or have marked 1 of the multiple `SpringSessionGemFireConfigurer` beans as primary by using Spring's `@Primary` context annotation.

3.4.1. Configuration Precedence

A `SpringSessionGemFireConfigurer` takes precedence over either the `@EnableGemFireHttpSession` annotation attributes or any of the well-known and documented Spring Session for Apache Geode properties (e.g. `spring.session.data.gemfire.session.expiration.max-inactive-interval-seconds`) defined in Spring Boot `application.properties`.

If more than 1 configuration approach is employed by your Web application, the following precedence will apply:

1. `SpringSessionGemFireConfigurer` "implemented" callback methods
2. Documented Spring Session for Apache Geode properties (See corresponding `@EnableGemFireHttpSession` annotation attribute Javadoc; e.g.

`spring.session.data.gemfire.session.region.name`)

3. `@EnableGemFireHttpSession` annotation attributes

Spring Session for Apache Geode is careful to only apply configuration from a `SpringSessionGemFireConfigurer` bean declared in the Spring container for the methods you have actually implemented.

In our example above, since you did not implement the `getRegionName()` method, the name of the Apache Geode Region managing the `HttpSession` state will not be determined by the Configurer.

Example

By way of example, consider the following configuration:

Example Spring Session for Apache Geode Configuration

```
@ClientCacheApplication
@EnableGemFireHttpSession(
    maxInactiveIntervalInSeconds = 3600,
    poolName = "DEFAULT"
)
class MySpringSessionConfiguration {

  @Bean
  SpringSessionGemFireConfigurer sessionExpirationTimeoutConfigurer() {

    return new SpringSessionGemFireConfigurer() {

      @Override
      public int getMaxInactiveIntervalInSeconds() {
        return 300;
      }
    };
  }
}
```

In addition, consider the following Spring Boot `application.properties` file:

1. Spring Boot `application.properties`

```
spring.session.data.gemfire.session.expiration.max-inactive-interval-seconds = 900
spring.session.data.gemfire.session.region.name = Sessions
```

The Session expiration timeout will be 300 seconds, or 5 minutes, overriding both the property (i.e. `spring.session.data.gemfire.session.expiration.max-inactive-interval-seconds`) of 900 seconds, or 15 minutes, as well as the explicit `@EnableGemFireHttpSession.maxInactiveIntervalInSeconds` annotation attribute value of 3600 seconds, or 1 hour.

Since the "sessionExpirationTimeoutConfigurer" bean does not override the `getRegionName()`

method, the Session Region name will be determined by the property (i.e. `spring.session.data.gemfire.session.region.name`), set to "Sessions", which overrides the implicit `@EnableGemFireHttpSession.regionName` annotation attribute's default value of "ClusteredSpringSessions".

The `@EnableGemFireHttpSession.poolName` annotation attribute's value of "DEFAULT" will determine the name of the Pool used when sending Region operations between the client and server to manage Session state on the server(s) since neither the corresponding property (i.e. spring.session.data.gemfire.cache.client.pool.name`) was set nor was the `SpringSessionGemFireConfigurer.getPoolName()` method overridden by the "sessionExpirationTimeoutConfigurer" bean.

And finally, the client Region used to manage Session state will have a data management policy of `PROXY`, the default value for the `@EnableGemFireHttpSession.clientRegionShortcut` annotation attribute, which was not explicitly set, nor was the corresponding property (i.e. `spring.session.data.gemfire.cache.client.region.shortcut`) for this attribute. And, because the `SpringSessionConfigurer.getClientRegionShortcut()` method was not overridden, the default value is used.

3.5. Apache Geode Expiration

By default, Apache Geode is configured with a Region Entry, Idle Timeout (TTI) Expiration Policy, using an expiration timeout of 30 minutes and INVALIDATE entry as the action. This means when a user's Session remains inactive (i.e. idle) for more than 30 minutes, the Session will expire and is invalidated, and the user must begin a new Session in order to continue to use the application.

However, what if you have application specific requirements around Session state management and expiration, and using the default, Idle Timeout (TTI) Expiration Policy is insufficient for your Use Case (UC)?

Now, Spring Session for Apache Geode supports application specific, custom expiration policies. As an application developer, you may specify custom rules governing the expiration of a Session managed by Spring Session, backed by Apache Geode.

Spring Session for Apache Geode provides the new `SessionExpirationPolicy` *Strategy* interface.

SessionExpirationPolicy interface

```
@FunctionalInterface
interface SessionExpirationPolicy {

    // determine timeout for expiration of individual Session
    Optional<Duration> determineExpirationTimeout(Session session);

    // define the action taken on expiration
    default ExpirationAction getExpirationAction() {
        return ExpirationAction.INVALIDATE;
    }

    enum ExpirationAction {

        DESTROY,
        INVALIDATE

    }
}
```

You implement this interface to specify the Session expiration policies required by your application and then register the instance as a bean in the Spring application context.

Use the `@EnableGemFireHttpSession` annotation, `sessionExpirationPolicyBeanName` attribute to configure the name of the `SessionExpirationPolicy` bean implementing your custom application policies and rules for Session expiration.

For example:

Custom `SessionExpirationPolicy`

```
class MySessionExpirationPolicy implements SessionExpirationPolicy {

    public Duration determineExpirationTimeout(Session session) {
        // return a java.time.Duration specifying the length of time until the Session
should expire
    }
}
```

Then, in your application class, simple declare the following:

Custom `SessionExpirationPolicy` *configuration*

```
@SpringBootApplication
@EnableGemFireHttpSession(
    maxInactiveIntervalInSeconds = 600,
    sessionExpirationPolicyBeanName = "expirationPolicy"
)
class MySpringSessionApplication {

    @Bean
    SessionExpirationPolicy expirationPolicy() {
        return new MySessionExpirationPolicy();
    }
}
```

Alternatively, the name of the `SessionExpirationPolicy` bean can be configured using the `spring.session.data.gemfire.session.expiration.bean-name` property, or by declaring a `SpringSessionGemFireConfigurer` bean in the Spring container and overriding the `getSessionExpirationPolicyBeanName()` method.

You are only required to implement the `determineExpirationTimeout(:Session):Optional<Duration>` method, which encapsulates the rules to determine when the Session should expire. The expiration timeout for a Session is expressed as an `Optional` of `java.time.Duration`, which specifies the length of time until the Session expires.

The `determineExpirationTimeout` method can be Session specific and may change with each invocation.

Optionally, you may implement the `getAction` method to specify the action taken when the Session expires. By default, the Region Entry (i.e. Session) is invalidated. Another option is to destroy the Region Entry on expiration, which removes both the key (Session ID) and value (Session). Invalidate only removes the value.

Under-the-hood, the `SessionExpirationPolicy` is adapted into an instance of the Apache Geode `CustomExpiry` interface. This Spring Session `CustomExpiry` object is then set as the Session Region's custom entry idle timeout expiration policy.

During expiration determination, the `CustomExpiry.getExpiry(:Region.Entry<String, Session>):ExpirationAttributes` method is invoked for each entry (i.e. Session) in the Region every time the expiration thread(s) run, which in turn calls our `SessionExpirationPolicy.determineExpirationTimout(:Session):Optional<Duration>` method. The returned `java.time.Duration` is converted to seconds and used as the expiration timeout in the `ExpirationAttributes` returned from the `CustomExpiry.getExpiry(..)` method invocation.

Apache Geode's expiration thread(s) run once every second, evaluating each entry (i.e. Session) in the Region to determine if the entry has expired. You can control the number of expiration threads with the `gemfire.EXPIRY_THREADS` property. See the Apache Geode docs for more details.

3.5.1. Expiration Timeout Configuration

If you would like to base the expiration timeout for your custom `SessionExpirationPolicy` on the `@EnableGemFireHttpSession` annotation, `maxInactiveIntervalInSeconds` attribute, or alternatively, the corresponding `spring.session.data.gemfire.session.expiration.max-inactive-interval-seconds` property, then your custom `SessionExpirationPolicy` implementation may also implement the `SessionExpirationTimeoutAware` interface.

The `SessionExpirationTimeoutAware` interface is defined as:

SessionExpirationTimeoutAware interface

```
interface SessionExpirationTimeoutAware {

    void setExpirationTimeout(Duration expirationTimeout);

}
```

When your custom `SessionExpirationPolicy` implementation also implements the `SessionExpirationTimeoutAware` interface, then Spring Session for Apache Geode will supply your implementation with the value from the `@EnableGemFireHttpSession` annotation, `maxInactiveIntervalInSeconds` attribute, or from the `spring.session.data.gemfire.session.expiration.max-inactive-interval-seconds` property if set, or from any `SpringSessionGemFireConfigurer` bean declared in the Spring application context, as an instance of `java.time.Duration`.

If more than 1 configuration option is used, the following order takes precedence:

1. `SpringSessionGemFireConfigurer.getMaxInactiveIntervalInSeconds()`
2. `spring.session.data.gemfire.session.expiration.max-inactive-interval-seconds` property
3. `@EnableGemFireHttpSession` annotation, `maxInactiveIntervalInSeconds` attribute

3.5.2. Fixed Timeout Expiration

For added convenience, Spring Session for Apache Geode provides an implementation of the `SessionExpirationPolicy` interface for fixed duration expiration (or "*Absolute session timeouts*" as described in core Spring Session Issue #922).

It is perhaps necessary, in certain cases, such as for security reasons, to expire the user's Session after a fixed length of time (e.g. every hour), regardless if the user's Session is still active.

Spring Session for Apache Geode provides the `FixedTimeoutSessionExpirationPolicy` implementation out-of-the-box for this exact Use Case (UC). In addition to handling fixed duration expiration, it is

also careful to still consider and apply the default, idle expiration timeout.

For instance, consider a scenario where a user logs in, beginning a Session, is active for 10 minutes and then leaves letting the Session sit idle. If the fixed duration expiration timeout is set for 60 minutes, but the idle expiration timeout is only set for 30 minutes, and the user does not return, then the Session should expire in 40 minutes and not 60 minutes when the fixed duration expiration would occur.

Conversely, if the user is busy for a full 40 minutes, thereby keeping the Session active, thus avoiding the 30 minute idle expiration timeout, and then leaves, then our fixed duration expiration timeout should kick in and expire the user's Session right at 60 minutes, even though the user's idle expiration timeout would not occur until 70 minutes in (40 min (active) + 30 min (idle) = 70 minutes).

Well, this is exactly what the `FixedTimeoutSessionExpirationPolicy` does.

To configure the `FixedTimeoutSessionExpirationPolicy`, do the following:

Fixed Duration Expiraton Configuration

```
@SpringBootApplication
@EnableGemFireHttpSession(sessionExpirationPolicyBeanName =
"fixedTimeoutExpirationPolicy")
class MySpringSessionApplication {

    @Bean
    SessionExpirationPolicy fixedTimeoutExpirationPolicy() {
        return new FixedTimeoutSessionExpirationPolicy(Duration.ofMinutes(60L));
    }
}
```

In the example above, the `FixedTimeoutSessionExpirationPolicy` was declared as a bean in the Spring application context and initialized with a fixed duration expiration timeout of 60 minutes. As a result, the users Session will either expire after the idle timeout (which defaults to 30 minutes) or after the fixed timeout (configured to 60 minutes), which ever occurs first.

It is also possible to implement lazy, fixed duration expiration timeout on Session access by using the Spring Session for Apache Geode `FixedDurationExpirationSessionRepositoryBeanPostProcessor`. This BPP wraps any data store specific `SessionRepository` in a `FixedDurationExpirationSessionRepository` implementation that evaluates a Sessions expiration on access, only. This approach is agnostic to the underlying data store and therefore can be used with any Spring Session provider. The expiration determination is based solely on the Session `creationTime` property and the required `java.time.Duration` specifying the fixed duration expiration timeout.

The `FixedDurationExpirationSessionRepository` should not be used in strict expiration timeout cases, such as when the Session must expire immediately after the fixed duration expiration timeout has elapsed. Additionally, unlike the `FixedTimeoutSessionExpirationPolicy`, the `FixedDurationExpirationSessionRepository` does not take idle expiration timeout into consideration. That is, it only uses the fixed duration when determining the expiration timeout for a given Session.

3.5.3. `SessionExpirationPolicy` Chaining

Using the Composite software design pattern, you can treat a group of `SessionExpirationPolicy` instances as a single instance, functioning as if in a chain much like the chain of Servlet Filters themselves.

The *Composite software design pattern* is a powerful pattern and is supported by the `SessionExpirationPolicy`, `@FunctionalInterface`, simply by returning an `Optional` of `java.time.Duration` from the `determineExpirationTimeout` method.

This allows each composed `SessionExpirationPolicy` to "optionally" return a `Duration` only if the expiration could be determined by this instance. Alternatively, this instance may punt to the next `SessionExpirationPolicy` in the composition, or chain until either a non-empty expiration timeout is returned, or ultimately no expiration timeout is returned.

In fact, this very policy is used internally by the `FixedTimeoutSessionExpirationPolicy`, which will return `Optional.empty()` in the case where the idle timeout will occur before the fixed timeout. By returning no expiration timeout, Apache Geode will defer to the default, configured entry idle timeout expiration policy on the Region managing Session state.

This exact behavior is also documented in the `org.apache.geode.cache.CustomExpiry.getExpiry(:Region.Entry<String, Session>):ExpirationAttributes` method.

3.6. Apache Geode Serialization

In order to transfer data between clients and servers, or when data is distributed and replicated between peer nodes in a cluster, the data must be serialized. In this case, the data in question is the Session's state.

Anytime a Session is persisted or accessed in a client/server topology, the Session's state is sent over-the-wire. Typically, a Spring Boot application with Spring Session enabled will be a client to the server(s) in a cluster of Apache Geode nodes.

On the server-side, Session state maybe distributed across several servers (data nodes) in the cluster to replicate the data and guarantee a high availability of the Session state. When using Apache Geode, data can be partitioned, or sharded, and a redundancy-level can be specified. When the data is distributed for replication, it must also be serialized to transfer the Session state among the peer nodes in the cluster.

Out-of-the-box, Apache Geode supports *Java Serialization*. There are many advantages to *Java Serialization*, such as handling cycles in the object graph, or being universally supported by any application written in Java. However, *Java Serialization* is very verbose and is not the most efficient over-the-wire format.

As such, Apache Geode provides its own serialization frameworks to serialize Java types:

1. Data Serialization
2. PDX Serialization

3.6.1. Apache Geode Serialization Background

As mentioned above, Apache Geode provide 2 additional serialization frameworks: *Data Serialization* and PDX *Serialization*.

Data Serialization

Data Serialization is a very efficient format (i.e. *fast* and *compact*), with little overhead when compared to *Java Serialization*.

It supports Delta Propagation by sending only the bits of data that actually changed as opposed to sending the entire object, which certainly cuts down on the amount of data sent over the network in addition to reducing the amount of IO when data is persisted or overflowed to disk.

However, *Data Serialization* incurs a CPU penalty anytime data is transferred over-the-wire, or persisted/overflowed to and accessed from disk, since the receiving end performs a deserialization. In fact, anytime *Delta Propagation* is used, the object must be deserialized on the receiving end in order to apply the "delta". Apache Geode applies deltas by invoking a method on the object that implements the `org.apache.geode.Delta` interface. Clearly, you cannot invoke a method on a serialized object.

PDX

PDX, on the other hand, which stands for *Portable Data Exchange*, retains the form in which the data was sent. For example, if a client sends data to a server in PDX format, the server will retain the data as PDX serialized bytes and store them in the cache `Region` for which the data access operation was targeted.

Additionally, PDX, as the name implies, is "*portable*", meaning it enables both Java and Native Language Clients, such as C, C++ and C# clients, to inter-operate on the same data set.

PDX even allows OQL queries to be performed on the serialized bytes without causing the objects to be deserialized first in order to evaluate the query predicate and execute the query. This can be accomplished since Apache Geode maintains a "*Type Registry*" containing type meta-data for the objects that get serialized and stored in Apache Geode using PDX.

However, portability does come with a cost, having slightly more overhead than *Data Serialization*. Still, PDX is far more efficient and flexible than *Java Serialization*, which stores type meta-data in the serialized bytes of the object rather than in a separate Type Registry as in Apache Geode's case when using PDX.

PDX does not support Deltas. Technically, a PDX serializable object can be used in *Delta Propagation* by implementing the `org.apache.geode.Delta` interface, and only the "delta" will be sent, even in the context of PDX. But then, the PDX serialized object must be deserialized to apply the delta. Remember, a method is invoked to apply the delta, which defeats the purpose of using PDX in the first place.

When developing Native Clients (e.g. C) that manage data in a {data-store-name} cluster, or even when mixing Native Clients with Java clients, typically there will not be any associated Java types provided on the classpath of the servers in the cluster. With PDX, it is not necessary to provide the Java types on the classpath, and many users who only develop and use Native Clients to access data stored in {data-store-name} will not provide any Java types for their corresponding C/C/C# types.

Apache Geode also support JSON serialized to/from PDX. In this case, it is very likely that Java types will not be provided on the servers classpath since many different languages (e.g. JavaScript, Python, Ruby) supporting JSON can be used with Apache Geode.

Still, even with PDX in play, users must take care not to cause a PDX serialized object on the servers in the cluster to be deserialized.

For example, consider an OQL query on an object of the following Java type serialized as PDX...

```
@Region("People")
class Person {

  private LocalDate birthDate;
  private String name;

  public int getAge() {
    // no explicit 'age' field/property in Person
    // age is just implemented in terms of the 'birthDate' field
  }
}
```

Subsequently, if the OQL query invokes a method on a `Person` object, such as:

```
SELECT * FROM /People p WHERE p.age >= 21
```

Then, this is going to cause a PDX serialized `Person` object to be deserialized since `age` is not a field of `Person`, but rather a method containing a computation based on another field of `Person` (i.e. `birthDate`). Likewise, calling any `java.lang.Object` method in a OQL query, like `Object.toString()`, is going to cause a deserialization to happen as well.

Apache Geode does provide the `read-serialized` configuration setting so that any cache `Region.get(key)` operations that are potentially invoked inside a `Function` does not cause PDX serialized objects to be deserialized. But, nothing will prevent an ill-conceived OQL query from causing a deserialization, so be careful.

Data Serialization + **PDX** + *Java Serialization*

It is possible for Apache Geode to support all 3 serialization formats simultaneously.

For instance, your application domain model might contain objects that implement the `java.io.Serialiable` interface, and you may be using a combination of the *Data Serialization* framework along with PDX.

While using *Java Serialization* with *Data Serialization* and PDX is possible, it is generally preferable and recommended to use 1 serialization strategy.

Unlike *Java Serialization*, *Data Serialization* and PDX *Serialization* do not handle object graph cycles.

More background on Apache Geode's serialization mechanics can be found here.

3.6.2. Serialization with Spring Session

Previously, Spring Session for Apache Geode only supported Apache Geode *Data Serialization* format. The main motivation behind this was to take advantage of Apache Geode's *Delta Propagation* functionality since a Session's state can be arbitrarily large.

However, as of Spring Session for Apache Geode 2.0, PDX is also supported and is now the new, default serialization option. The default was changed to PDX in Spring Session for Apache Geode 2.0 primarily because PDX is the most widely used and requested format by users.

PDX is certainly the most flexible format, so much so that you do not even need Spring Session for Apache Geode or any of its transitive dependencies on the classpath of the servers in the cluster to use Spring Session with Apache Geode. In fact, with PDX, you do not even need to put your application domain object types stored in the (HTTP) Session on the servers' classpath either.

Essentially, when using PDX serialization, Apache Geode does not require the associated Java types to be present on the servers' classpath. So long as no deserialization happens on the servers in the cluster, you are safe.

The `@EnableGemFireHttpSession` annotation introduces the **new** `sessionSerializerBeanName` attribute that a user can use to configure the name of a bean declared and registered in the Spring container implementing the desired serialization strategy. The serialization strategy is used by Spring Session for Apache Geode to serialize the Session state.

Out-of-the-box, Spring Session for Apache Geode provides 2 serialization strategies: 1 for PDX and 1 for *Data Serialization*. It automatically registers both serialization strategy beans in the Spring container. However, only 1 of those strategies is actually used at runtime, PDX!

The 2 beans registered in the Spring container implementing *Data Serialization* and PDX are named `SessionDataSerializer` and `SessionPdxSerializer`, respectively. By default, the `sessionSerializerBeanName` attribute is set to `SessionPdxSerializer`, as if the user annotated his/her Spring Boot, Spring Session enabled application configuration class with:

```
@SpringBootApplication
@EnableGemFireHttpSession(sessionSerializerBeanName = "SessionPdxSerializer")
class MySpringSessionApplication {  }
```

It is a simple matter to change the serialization strategy to *Data Serialization* instead by setting the `sessionSerializerBeanName` attribute to `SessionDataSerializer`, as follows:

```
@SpringBootApplication
@EnableGemFireHttpSession(sessionSerializerBeanName = "SessionDataSerializer")
class MySpringSessionApplication {  }
```

Since these 2 values are so common, Spring Session for Apache Geode provides constants for each value in the `GemFireHttpSessionConfiguration` class: `GemFireHttpSessionConfiguration.SESSION_PDX_SERIALIZER_BEAN_NAME` and `GemFireHttpSessionConfiguration.SESSION_DATA_SERIALIZER_BEAN_NAME`. So, you could explicitly configure PDX, as follows:

```
import
org.springframework.session.data.geode.config.annotation.web.http.GemFireHttpSessionCo
nfiguration;

@SpringBootApplication
@EnableGemFireHttpSession(sessionSerializerBeanName =
GemFireHttpSessionConfiguration.SESSION_PDX_SERIALIZER_BEAN_NAME)
class MySpringSessionApplication {  }
```

With 1 attribute and 2 provided bean definitions out-of-the-box, you can specify which Serialization framework you wish to use with your Spring Boot, Spring Session enabled application backed by Apache Geode.

3.6.3. Spring Session for Apache Geode Serialization Framework

To abstract away the details of Apache Geode's *Data Serialization* and PDX *Serialization* frameworks, Spring Session for Apache Geode provides its own Serialization framework (facade) wrapping Apache Geode's Serialization frameworks.

The Serialization API exists under the `org.springframework.session.data.gemfire.serialization` package. The primary interface in this API is the `org.springframework.session.data.gemfire.serialization.SessionSerializer`.

The interface is defined as:

Spring Session `SessionSerializer` *interface*

```
interface SessionSerializer<T, IN, OUT> {

  void serialize(T session, OUT out);

  T deserialize(IN in);

  boolean canSerialize(Class<?> type);

  boolean canSerialize(Object obj) {
    // calls Object.getClass() in a null-safe way and then calls and returns
canSerialize(:Class)
  }
}
```

Basically, the interface allows you to serialize and deserialize a Spring `Session` object.

The `IN` and `OUT` type parameters and corresponding method arguments of those types provide reference to the objects responsible for writing the `Session` to a stream of bytes or reading the `Session` from a stream of bytes. The actual arguments will be type specific, based on the underlying Apache Geode Serialization strategy configured.

For instance, when using Apache Geode's PDX *Serialization* framework, `IN` and `OUT` will be instances of `org.apache.geode.pdx.PdxReader` and `org.apache.geode.pdx.PdxWriter`, respectively. When Apache Geode's *Data Serialization* framework has been configured, then `IN` and `OUT` will be instances of `java.io.DataInput` and `java.io.DataOutput`, respectively.

These arguments are provided to the `SessionSerializer` implementation by the framework automatically, and as previously mentioned, is based on the underlying Apache Geode Serialization strategy configured.

Essentially, even though Spring Session for Apache Geode provides a facade around Apache Geode's Serialization frameworks, under-the-hood Apache Geode still expects one of these Serialization frameworks is being used to serialize data to/from Apache Geode.

So what purpose does the `SessionSerializer` *interface really serve then?*

Effectively, it allows a user to customize what aspects of the Session's state actually gets serialized and stored in Apache Geode. Application developers can provide their own custom, application-specific `SessionSerializer` implementation, register it as a bean in the Spring container, and then configure it to be used by Spring Session for Apache Geode to serialize the Session state, as follows:

```
@EnableGemFireHttpSession(sessionSerializerBeanName = "MyCustomSessionSerializer")
class MySpringSessionDataGemFireApplication {

  @Bean("MyCustomSessionSerializer")
  SessionSerializer<Session, ?, ?> myCustomSessionSerializer() {
    // ...
  }
}
```

Implementing a SessionSerializer

Spring Session for Apache Geode provides assistance when a user wants to implement a custom `SessionSerializer` that fits into one of Apache Geode's Serialization frameworks.

If the user just implements the `org.springframework.session.data.gemfire.serialization.SessionSerializer` interface directly without extending from one of Spring Session for Apache Geode's provided abstract base classes, related to 1 of Apache Geode's Serialization frameworks , then Spring Session for Apache Geode will wrap the user's custom `SessionSerializer` implementation in an instance of `org.springframework.session.data.gemfire.serialization.pdx.support.PdxSerializerSessionSerializerAdapter` and register it with Apache Geode as a `org.apache.geode.pdx.PdxSerializer`.

Spring Session for Apache Geode is careful not to stomp on any existing `PdxSerializer` implementation that a user may have already registered with Apache Geode by some other means. Indeed, several different, provided implementations of Apache Geode's `org.apache.geode.pdx.PdxSerializer` interface exists:

- Apache Geode itself provides the `org.apache.geode.pdx.ReflectionBasedAutoSerializer`.
- Spring Data for Apache Geode (SDG) provides the `org.springframework.data.gemfire.mapping.MappingPdxSerializer`, which is used in the SD *Repository* abstraction and SDG's extension to handle mapping PDX serialized types to the application domain object types defined in the application *Repository* interfaces.

This is accomplished by obtaining any currently registered `PdxSerializer` instance on the cache and composing it with the `PdxSerializerSessionSerializerAdapter` wrapping the user's custom application `SessionSerializer` implementation and re-registering this "*composite*" `PdxSerializer` on the Apache Geode cache. The "*composite*" `PdxSerializer` implementation is provided by Spring Session for Apache Geode's `org.springframework.session.data.gemfire.pdx.support.ComposablePdxSerializer` class when entities are stored in Apache Geode as PDX.

If no other `PdxSerializer` was currently registered with the Apache Geode cache, then the adapter is simply registered.

Of course, you are allowed to force the underlying Apache Geode Serialization strategy used with a custom `SessionSerializer` implementation by doing 1 of the following:

1. The custom `SessionSerializer` implementation can implement Apache Geode's `org.apache.geode.pdx.PdxSerializer` interface, or for convenience, extend Spring Session for

Apache Geode's `org.springframework.session.data.gemfire.serialization.pdx.AbstractPdxSerializableSessionSerializer` class and Spring Session for Apache Geode will register the custom `SessionSerializer` as a `PdxSerializer` with Apache Geode.

2. The custom `SessionSerializer` implementation can extend the Apache Geode's `org.apache.geode.DataSerializer` class, or for convenience, extend Spring Session for Apache Geode's `org.springframework.session.data.gemfire.serialization.data.AbstractDataSerializableSessionSerializer` class and Spring Session for Apache Geode will register the custom `SessionSerializer` as a `DataSerializer` with Apache Geode.

3. Finally, a user can create a custom `SessionSerializer` implementation as before, not specifying which Apache Geode Serialization framework to use because the custom `SessionSeriaizer` implementation does not implement any Apache Geode serialization interfaces or extend from any of Spring Session for Apache Geode's provided abstract base classes, and still have it registered in Apache Geode as a `DataSerializer` by declaring an additional Spring Session for Apache Geode bean in the Spring container of type `org.springframework.session.data.gemfire.serialization.data.support.DataSerializerSessionSerializerAdapter`, like so...

Forcing the registration of a custom SessionSerializer as a DataSerializer in Apache Geode

```
@EnableGemFireHttpSession(sessionSerializerBeanName = "customSessionSerializer")
class Application {

    @Bean
    DataSerializerSessionSerializerAdapter dataSerializerSessionSerializer() {
        return new DataSerializerSessionSerializerAdapter();
    }

    @Bean
    SessionSerializer<Session, ?, ?> customSessionSerializer() {
        // ...
    }
}
```

Just by the very presence of the `DataSerializerSessionSerializerAdapter` registered as a bean in the Spring container, any neutral custom `SessionSerializer` implementation will be treated and registered as a `DataSerializer` in Apache Geode.

Additional Support for Data Serialization

Please feel free to skip this section if you are configuring and bootstraping Apache Geode servers in your cluster using Spring (Boot) since generally, the information that follows will not apply. Of course, it all depends on your declared dependencies and your Spring configuration. However, if you are using ***Gfsh*** to start the servers in your cluster, then definitely read on.

Background

When using Apache Geode's *DataSerialization* framework, especially from the client when serializing (HTTP) Session state to the servers in the cluster, you must take care to configure the Apache Geode servers in your cluster with the appropriate dependencies. This is especially true when leveraging deltas as explained in the earlier section on *Data Serialization*.

When using the *DataSerialization* framework as your serialization strategy to serialize (HTTP) Session state from your Web application clients to the servers, then the servers must be properly configured with the Spring Session for Apache Geode class types used to represent the (HTTP) Session and its contents. This means including the Spring JARs on the servers classpath.

Additionally, using *DataSerialization* may also require you to include the JARs containing your application domain classes that are used by your Web application and put into the (HTTP) Session as Session Attribute values, particularly if:

1. Your types implement the `org.apache.geode.DataSerializable` interface.
2. Your types implement the `org.apache.geode.Delta` interface.
3. You have registered a `org.apache.geode.DataSerializer` that identifies and serializes the types.
4. Your types implement the `java.io.Serializable` interface.

Of course, you must ensure your application domain object types put in the (HTTP) Session are serializable in some form or another. However, you are not strictly required to use *DataSerialization* nor are you necessarily required to have your application domain object types on the servers classpath if:

1. Your types implement the `org.apache.geode.pdx.PdxSerializable` interface.
2. Or, you have registered an `org.apache.geode.pdx.PdxSerializer` that properly identifies and serializes your application domain object types.

Apache Geode will apply the following order of precedence when determining the serialization strategy to use to serialize an object graph:

1. First, `DataSerializable` objects and/or any registered `DataSerializers` identifying the objects to serialize.
2. Then `PdxSerializable` objects and/or any registered `PdxSerializer` identifying the objects to serialize.
3. And finally, all `java.io.Serializable` types.

This also means that if a particular application domain object type (e.g. `A`) implements `java.io.Serializable`, however, a (custom) `PdxSerializer` has been registered with Apache Geode identifying the same application domain object type (i.e. `A`), then Apache Geode will use PDX to serialize "A" and **not** Java Serialization, in this case.

This is especially useful since then you can use *DataSerialization* to serialize the (HTTP) Session object, leveraging Deltas and all the powerful features of *DataSerialization*, but then use PDX to serialize your application domain object types, which greatly simplifies the configuration and/or effort involved.

Now that we have a general understanding of why this support exists, how do you enable it?

Configuration

First, create an Apache Geode `cache.xml`, as follows:

Apache Geode `cache.xml` configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<cache xmlns="http://geode.apache.org/schema/cache"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://geode.apache.org/schema/cache
https://geode.apache.org/schema/cache/cache-1.0.xsd"
       version="1.0">

  <initializer>
    <class-name>

org.springframework.session.data.gemfire.serialization.data.support.DataSerializableSe
ssionSerializerInitializer
    </class-name>
  </initializer>

</cache>
```

Then, start your servers in *Gfsh* using:

Starting Server with Gfsh

```
gfsh> start server --name=InitializedServer --cache-xml-file=/path/to/cache.xml
--classpath=...
```

Configuring the Apache Geode server `classpath` with the appropriate dependencies is the tricky part, but generally, the following should work:

CLASSPATH configuration

```
gfsh> set variable --name=REPO_HOME --value=${USER_HOME}/.m2/repository

gfsh> start server ... --classpath=\
${REPO_HOME}/org/springframework/spring-core/{spring-version}/spring-core-{spring-
version}.jar\
:${REPO_HOME}/org/springframework/spring-aop/{spring-version}/spring-aop-{spring-
version}.jar\
:${REPO_HOME}/org/springframework/spring-beans/{spring-version}/spring-beans-{spring-
version}.jar\
:${REPO_HOME}/org/springframework/spring-context/{spring-version}/spring-context-
{spring-version}.jar\
:${REPO_HOME}/org/springframework/spring-context-support/{spring-version}/spring-
context-support-{spring-version}.jar\
:${REPO_HOME}/org/springframework/spring-expression/{spring-version}/spring-
expression-{spring-version}.jar\
:${REPO_HOME}/org/springframework/spring-jcl/{spring-version}/spring-jcl-{spring-
version}.jar\
:${REPO_HOME}/org/springframework/spring-tx/{spring-version}/spring-tx-{spring-
version}.jar\
:${REPO_HOME}/org/springframework/data/spring-data-commons/{spring-data-commons-
version}/spring-data-commons-{spring-data-commons-version}.jar\
:${REPO_HOME}/org/springframework/data/spring-data-geode/{spring-data-geode-
version}/spring-data-geode-{spring-data-geode-version}.jar\
:${REPO_HOME}/org/springframework/session/spring-session-core/{spring-session-core-
version}/spring-session-core-{spring-session-core-version}.jar\
:${REPO_HOME}/org/springframework/session/spring-session-data-geode/{spring-session-
data-geode-version}/spring-session-data-geode-{spring-session-data-geode-version}.jar\
:${REPO_HOME}/org/slf4j/slf4j-api/1.7.25/slf4j-api-1.7.25.jar
```

Keep in mind, you may need to add your application domain object JAR files to the server classpath as well.

To get a complete picture of how this works, see the sample.

Customizing Change Detection

By default, anytime the Session is modified (e.g. the `lastAccessedTime` is updated to the current time), the Session is considered dirty by Spring Session for Apache Geode (SSDG). When using Apache Geode *Data Serialization* framework, it is extremely useful and valuable to take advantage of Apache Geode's Delta Propagation capabilities as well.

When using *Data Serialization*, SSDG also uses *Delta Propagation* to send only changes to the Session state between the client and server. This includes any Session attributes that may have been added, removed or updated.

By default, anytime `Session.setAttribute(name, value)` is called, the Session attribute is considered "dirty" and will be sent in the delta between the client and server. This is true even if your application domain object has not been changed.

Typically, there is never a reason to call `Session.setAttribute(..)` unless your object has been changed. However, if this can occur, and your objects are relatively large (with a complex object hierarchy), then you may want to consider either:

1. Implementing the Delta interface in your application domain object model, while useful, is very invasive, or…
2. Provide a custom implementation of SSDG's `org.springframework.session.data.gemfire.support.IsDirtyPredicate` strategy interface.

Out of the box, SSDG provides 5 implementations of the `IsDirtyPredicate` strategy interface:

Table 5. `IsDirtyPredicate` implementations

Class	Description	Default
`IsDirtyPredicate.ALWAYS_DIRTY`	New Session attribute values are always considered dirty.	
`IsDirtyPredicate.NEVER_DIRTY`	New Session attribute values are never considered dirty.	
`DeltaAwareDirtyPredicate`	New Session attribute values are considered dirty when the old value and new value are different, if the new value's type does not implement `Delta` or the new value's `Delta.hasDelta()` method returns **true**.	Yes
`EqualsDirtyPredicate`	New Session attribute values are considered dirty iff the old value is not equal to the new value as determined by `Object.equals(:Object)` method.	
`IdentityEqualsPredicate`	New Session attributes values are considered dirty iff the old value is not the same as the new value using the identity equals operator (i.e. `oldValue != newValue`).	

As shown in the table above, the `DeltaAwareDirtyPredicate` is the **default** implementation used by SSDG. The `DeltaAwareDirtyPredicate` automatically takes into consideration application domain objects that implement the Apache Geode `Delta` interface. However, `DeltaAwareDirtyPredicate` works even when your application domain objects do not implement the `Delta` interface. SSDG will consider your application domain object to be dirty anytime the `Session.setAttribute(name, newValue)` is called providing the new value is not the same as old value, or the new value does not implement the `Delta` interface.

You can change SSDG's dirty implementation, determination strategy simply by declaring a bean in the Spring container of the `IsDirtyPredicate` interface type:

Overriding SSDG's default `IsDirtyPredicate` strategy

```
@EnableGemFireHttpSession
class ApplicationConfiguration {

  @Bean
  IsDirtyPredicate equalsDirtyPredicate() {
    return EqualsDirtyPredicate.INSTANCE;
  }
}
```

Composition

The `IsDirtyPredicate` interface also provides the `andThen(:IsDirtyPredicate)` and `orThen(:IsDirtyPredicate)` methods to compose 2 or more `IsDirtyPredicate` implementations in a composition in order to organize complex logic and rules for determining whether an application domain object is dirty or not.

For instance, you could compose both `EqualsDirtyPredicate` and `DeltaAwareDirtyPredicate` using the OR operator:

Composing `EqualsDirtyPredicate` with `DeltaAwareDirtyPredicate` using the logical OR operator

```
@EnableGemFireHttpSession
class ApplicationConfiguration {

  @Bean
  IsDirtyPredicate equalsOrThenDeltaDirtyPredicate() {

    return EqualsDirtyPredicate.INSTANCE
      .orThen(DeltaAwareDirtyPredicate.INSTANCE);
  }
}
```

You may even implement your own, custom `IsDirtyPredicates` based on specific application domain object types:

Application Domain Object Type-specific `IsDirtyPredicate` implementations

```
class CustomerDirtyPredicate implements IsDirtyPredicate {

  public boolean isDirty(Object oldCustomer, Object newCustomer) {

      if (newCustomer instanceof Customer) {
        // custom logic to determine if a new Customer is dirty
      }

      return true;
  }
}

class AccountDirtyPredicate implements IsDirtyPredicate {

  public boolean isDirty(Object oldAccount, Object newAccount) {

      if (newAccount instanceof Account) {
        // custom logic to determine if a new Account is dirty
      }

      return true;
  }
}
```

Then combine `CustomerDirtyPredicate` with the `AccountDirtyPredicate` and a default predicate for fallback, as follows:

Composed and configured type-specific `IsDirtyPredicates`

```
@EnableGemFireHttpSession
class ApplicationConfiguration {

  @Bean
  IsDirtyPredicate typeSpecificDirtyPredicate() {

    return new CustomerDirtyPredicate()
      .andThen(new AccountDirtyPredicate())
      .andThen(IsDirtyPredicate.ALWAYS_DIRTY);
  }
}
```

The combinations and possibilities are endless.

Use caution when implementing custom `IsDirtyPredicate` strategies. If you incorrectly determine that your application domain object is not dirty when it actually is, then it will not be sent in the Session delta from the client to the server.

Changing the Session Representation

Internally, Spring Session for Apache Geode maintains 2 representations of the (HTTP) Session and the Session's attributes. Each representation is based on whether Apache Geode "*Deltas*" are supported or not.

Apache Geode *Delta Propagation* is only enabled by Spring Session for Apache Geode when using *Data Serialization* for reasons that were discussed earlier.

Effectively, the strategy is:

1. If Apache Geode *Data Serialization* is configured, then *Deltas* are supported and the `DeltaCapableGemFireSession` and `DeltaCapableGemFireSessionAttributes` representations are used.
2. If Apache Geode PDX *Serialization* is configured, then *Delta Propagation* will be disabled and the `GemFireSession` and `GemFireSessionAttributes` representations are used.

It is possible to override these internal representations used by Spring Session for Apache Geode, and for users to provide their own Session related types. The only strict requirement is that the Session implementation must implement the core Spring Session `org.springframework.session.Session` interface.

By way of example, let's say you want to define your own Session implementation.

First, you define the `Session` type. Perhaps your custom `Session` type even encapsulates and handles the Session attributes without having to define a separate type.

User-defined Session interface implementation

```
class MySession implements org.springframework.session.Session {
  // ...
}
```

Then, you would need to extend the `org.springframework.session.data.gemfire.GemFireOperationsSessionRepository` class and override the `createSession()` method to create instances of your custom `Session` implementation class.

Custom SessionRepository implementation creating and returning instances of the custom Session type

```
class MySessionRepository extends GemFireOperationsSessionRepository {

  @Override
  public Session createSession() {
    return new MySession();
  }
}
```

If you provide your own custom `SessionSerializer` implementation and Apache Geode PDX *Serialization* is configured, then you done.

However, if you configured Apache Geode *Data Serialization* then you must additionally provide a

custom implementation of the `SessionSerializer` interface and either have it directly extend Apache Geode's `org.apache.geode.DataSerializer` class, or extend Spring Session for Apache Geode's `org.springframework.session.data.gemfire.serialization.data.AbstractDataSerializableSessionSerializer` class and override the `getSupportedClasses():Class<?>[]` method.

For example:

Custom SessionSerializer for custom Session type

```
class MySessionSerializer extends AbstractDataSerializableSessionSerializer {

  @Override
  public Class<?>[] getSupportedClasses() {
    return new Class[] { MySession.class };
  }
}
```

Unfortunately, `getSupportedClasses()` cannot return the generic Spring Session `org.springframework.session.Session` interface type. If it could then we could avoid the explicit need to override the `getSupportedClasses()` method on the custom `DataSerializer` implementation. But, Apache Geode's *Data Serialization* framework can only match on exact class types since it incorrectly and internally stores and refers to the class type by name, which then requires the user to override and implement the `getSupportedClasses()` method.

3.7. How HttpSession Integration Works

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

This section describes how Spring Session provides transparent integration with `javax.servlet.http.HttpSession`. The intent is so users understand what is happening under-the-hood. This functionality is already implemented and integrated so you do not need to implement this logic yourself.

First, we create a custom `javax.servlet.http.HttpServletRequest` that returns a custom implementation of `javax.servlet.http.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 `javax.servlet.http.HttpSession` is overridden. All other methods are implemented by `javax.servlet.http.HttpServletRequestWrapper` and simply delegate to the original `javax.servlet.http.HttpServletRequest` implementation.

We replace the `javax.servlet.http.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 `javax.servlet.http.HttpServletRequest` implementation into the `FilterChain` we ensure that anything invoked after our `Filter` uses the custom `javax.servlet.http.HttpSession` implementation.

This highlights why it is important that Spring Session's `SessionRepositoryFilter` must be placed before anything that interacts with the `javax.servlet.http.HttpSession`.

3.8. HttpSessionListener

Spring Session supports `HttpSessionListener` by translating `SessionCreatedEvent` and `SessionDestroyedEvent` into `HttpSessionEvent` by declaring `SessionEventHttpSessionListenerAdapter`.

To use this support, you need to:

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

If you are using the configuration support documented in HttpSession with Apache Geode, 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`, then you can simply add `HttpSessionEventPublisher` as a bean.

3.9. Session

A `Session` is a simplified `Map` of key/value pairs with support for expiration.

3.10. SessionRepository

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

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

3.11. FindByIndexNameSessionRepository

Spring Session's most basic API for using a `Session` is the `SessionRepository`. The 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 Session's for Apache Geode 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.

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`.

3.12. EnableSpringHttpSession

The `@EnableSpringHttpSession` annotation can be added to any `@Configuration` class to expose the `SessionRepositoryFilter` as a bean in the Spring container named, "springSessionRepositoryFilter".

In order to leverage the annotation, a single `SessionRepository` bean must be provided.

3.13. EnableGemFireHttpSession

The `@EnableGemFireHttpSession` annotation can be added to any `@Configuration` class in place of the `@EnableSpringHttpSession` annotation to expose the `SessionRepositoryFilter` as a bean in the Spring container named, "springSessionRepositoryFilter" and to position Apache Geode as a provider managing `javax.servlet.http.HttpSession` state.

When using the `@EnableGemFireHttpSession` annotation, additional configuration is imported out-of-the-box that also provides a Apache Geode specific implementation of the `SessionRepository` interface named, `GemFireOperationsSessionRepository`.

3.14. GemFireOperationsSessionRepository

`GemFireOperationsSessionRepository` is a `SessionRepository` implementation that is implemented using Spring Session for Apache Geode's_ `GemFireOperationsSessionRepository`.

In a web environment, this repository is used in conjunction with the `SessionRepositoryFilter`.

This implementation supports `SessionCreatedEvents`, `SessionDeletedEvents` and `SessionDestroyedEvents` through `SessionEventHttpSessionListenerAdapter`.

3.14.1. Using Indexes with Apache Geode

While best practices concerning the proper definition of Indexes that positively impact Apache Geode's performance is beyond the scope of this document, it is important to realize that Spring Session for Apache Geode creates and uses Indexes to query and find Sessions efficiently.

Out-of-the-box, Spring Session for Apache Geode creates 1 Hash-typed Index on the principal name. There are two different built-in strategies for finding the principal name. The first strategy is that the value of the Session attribute with the name `FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME` will be Indexed to the same index name.

For example:

```
String indexName = FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME;

session.setAttribute(indexName, username);
Map<String, Session> idToSessions =
    this.sessionRepository.findByIndexNameAndIndexValue(indexName, username);
```

3.14.2. Using Indexes with Apache Geode & Spring Security

Alternatively, Spring Session for Apache Geode will map Spring Security's current `Authentication#getName()` to the Index `FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME`.

For example, if you are using Spring Security you can find the current user's sessions using:

```
SecurityContext securityContext = SecurityContextHolder.getContext();
Authentication authentication = securityContext.getAuthentication();
String indexName = FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME;

Map<String, Session> idToSessions =
    this.sessionRepository.findByIndexNameAndIndexValue(indexName,
authentication.getName());
```

3.14.3. Using Custom Indexes with Apache Geode

This enables developers using the `GemFireOperationsSessionRepository` programmatically to query and find all Sessions with a given principal name efficiently.

Additionally, Spring Session for Apache Geode will create a Range-based Index on the implementing Session's Map-type `attributes` property (i.e. on any arbitrary Session attribute) when a developer identifies 1 or more named Session attributes that should be indexed by Apache Geode.

Sessions attributes to index can be specified with the `indexableSessionAttributes` attribute on the `@EnableGemFireHttpSession` annotation. A developer adds this annotation to their Spring application `@Configuration` class when s/he wishes to enable Spring Session's support for `HttpSession` backed by Apache Geode.

```
String indexName = "name1";

session.setAttribute(indexName, indexValue);
Map<String, Session> idToSessions =
    this.sessionRepository.findByIndexNameAndIndexValue(indexName, indexValue);
```

Only Session attribute names identified in the `@EnableGemFireHttpSession` annotation's `indexableSessionAttributes` attribute will have an Index defined. All other Session attributes will not be indexed.

However, there is one catch. Any values stored in an indexable Session attributes must implement

the `java.lang.Comparable<T>` interface. If those object values do not implement `Comparable`, then Apache Geode will throw an error on startup when the Index is defined for Regions with persistent Session data, or when an attempt is made at runtime to assign the indexable Session attribute a value that is not `Comparable` and the Session is saved to Apache Geode.

Any Session attribute that is not indexed may store non-`Comparable` values.

To learn more about Apache Geode's Range-based Indexes, see Creating Indexes on Map Fields.

To learn more about Apache Geode Indexing in general, see Working with Indexes.

Chapter 4. Spring Session Community

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

4.1. 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*.

4.2. Source Code

The source code can be found on GitHub at https://github.com/spring-projects/spring-session-data-geode

4.3. Issue Tracking

We track issues in GitHub Issues at https://github.com/spring-projects/spring-session-data-geode/issues

4.4. Contributing

We appreciate Pull Requests.

4.5. License

Spring Session for Apache Geode and Spring Session for Pivotal GemFire are Open Source Software released under the Apache 2.0 license.

Chapter 5. Minimum Requirements

The minimum requirements for Spring Session are:

- Java 8+
- If you are running in a Servlet container (not required), Servlet 2.5+
- If you are using other Spring libraries (not required), the minimum required version is Spring Framework 5.0.x.
- `@EnableGemFireHttpSession` requires Spring Data for Apache Geode 2.0.x and Spring Data for Pivotal GemFire 2.0.x.
- `@EnableGemFireHttpSession` requires Apache Geode 1.2.x or Pivotal GemFire 9.1.x.

At its core Spring Session only has a required dependency on `spring-jcl`.