Session Attributes in Spring MVC

Last modified: September 11, 2022

1. Overview

When developing web applications, we often need to refer to the same attributes in several views. For example, we may have shopping cart contents that need to be displayed on multiple pages.

A good location to store those attributes is in the user's session.

In this tutorial, we'll focus on a simple example and **examine 2 different strategies for working with a session attribute**:

- **Using a scoped proxy**
- **Using the @*SessionAttributes* annotation**

2. Maven Setup

We'll use Spring Boot starters to bootstrap our project and bring in all necessary dependencies.

Our setup requires a parent declaration, web starter, and thymeleaf starter.

We'll also include the spring test starter to provide some additional utility in our unit tests:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.2</version>
    <relativePath/>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
     </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

The most recent versions of these dependencies can be found on Maven Central (https://search.maven.org/classic/#search%7Cga%7C1%7Cg%3A%22org.springframework.boot%22%20AND%20(a%3A%22spring-boot-starter-web%22%20OR%20a%3A%22spring-boot-starter-thymeleaf%22%20OR%20a%3A%22spring-boot-starter-test%22)).

3. Example Use Case

Our example will implement a simple "TODO" application. We'll have a form for creating instances of *TodoItem* and a list view that displays all *TodoItem*s.

If we create a *TodoItem* using the form, subsequent accesses of the form will be prepopulated with the values of the most recently added *TodoItem*. **We'll use this feature to demonstrate how to "remember" form values** that are stored in session scope.

Our 2 model classes are implemented as simple POJOs:

```
public class TodoItem {

    private String description;
    private LocalDateTime createDate;

    // getters and setters
}
```

```
public class TodoList extends ArrayDeque<TodoItem>{

}
```

Our *TodoList* class extends *ArrayDeque* to give us convenient access to the most recently added item via the *peekLast* method.

We'll need 2 controller classes: 1 for each of the strategies we'll look at. They'll have subtle differences but the core functionality will be represented in both. Each will have 3 *@RequestMapping*s:

- ***@GetMapping("/form")*** – This method will be responsible for initializing the form and rendering the form view. The method will prepopulate the form with the most recently added *TodoItem* if the *TodoList* is not empty.
- ***@PostMapping("/form")*** – This method will be responsible for adding the submitted *TodoItem* to the *TodoList* and redirecting to the list URL.
- ***@GetMapping("/todos.html")* –** This method will simply add the *TodoList* to the *Model* for display and render the list view.

4. Using a Scoped Proxy

4.1. Setup

In this setup, our *TodoList* is configured as a session-scoped *@Bean* that is backed by a proxy. The fact that the *@Bean* is a proxy means that we are able to inject it into our singleton-scoped *@Controller*.

Since there is no session when the context initializes, Spring will create a proxy of *TodoList* to inject as a dependency. The target instance of *TodoList* will be instantiated as needed when required by requests.

For a more in-depth discussion of bean scopes in Spring, refer to our article on the topic (/spring-bean-scopes).

First, we define our bean within a *@Configuration* class:

```
@Bean
@Scope(
  value = WebApplicationContext.SCOPE_SESSION,
  proxyMode = ScopedProxyMode.TARGET_CLASS)
public TodoList todos() {
    return new TodoList();
}
```

Next, we declare the bean as a dependency for the *@Controller* and inject it just as we would any other dependency:

```
@Controller
@RequestMapping("/scopedproxy")
public class TodoControllerWithScopedProxy {

    private TodoList todos;

    // constructor and request mappings
}
```

Finally, using the bean in a request simply involves calling its methods:

```
@GetMapping("/form")
public String showForm(Model model) {
    if (!todos.isEmpty()) {
        model.addAttribute("todo", todos.peekLast());
    } else {
        model.addAttribute("todo", new TodoItem());
    }
    return "scopedproxyform";
}
```

4.2. Unit Testing

In order to test our implementation using the scoped proxy, **we first configure a *SimpleThreadScope***. This will ensure that our unit tests accurately simulate runtime conditions of the code we are testing.

First, we define a *TestConfig* and a *CustomScopeConfigurer*.

```
@Configuration
public class TestConfig {

    @Bean
    public CustomScopeConfigurer customScopeConfigurer() {
        CustomScopeConfigurer configurer = new CustomScopeConfigurer();
        configurer.addScope("session", new SimpleThreadScope());
        return configurer;
    }
}
```

Now we can start by testing that an initial request of the form contains an uninitialized *TodoItem:*

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
@Import(TestConfig.class)
public class TodoControllerWithScopedProxyIntegrationTest {

    // ...

    @Test
    public void whenFirstRequest_thenContainsUnintializedTodo() throws
Exception {
        MvcResult result = mockMvc.perform(get("/scopedproxy/form"))
          .andExpect(status().isOk())
          .andExpect(model().attributeExists("todo"))
          .andReturn();

        TodoItem item = (TodoItem)
result.getModelAndView().getModel().get("todo");

        assertTrue(StringUtils.isEmpty(item.getDescription()));
    }
}
```

We can also confirm that our submit issues a redirect and that a subsequent form request is prepopulated with the newly added *TodoItem*:

```
@Test
public void
whenSubmit_thenSubsequentFormRequestContainsMostRecentTodo() throws
Exception {
    mockMvc.perform(post("/scopedproxy/form")
      .param("description", "newtodo"))
      .andExpect(status().is3xxRedirection())
      .andReturn();

    MvcResult result = mockMvc.perform(get("/scopedproxy/form"))
      .andExpect(status().isOk())
      .andExpect(model().attributeExists("todo"))
      .andReturn();
    TodoItem item = (TodoItem)
result.getModelAndView().getModel().get("todo");

    assertEquals("newtodo", item.getDescription());
}
```

4.3. Discussion

A key feature of using the scoped proxy strategy is that **it has no impact on request mapping method signatures.** This keeps readability on a very high level compared to the *@SessionAttributes* strategy.

It can be helpful to recall that controllers have *singleton* scope by default.

This is the reason why we must use a proxy instead of simply injecting a non-proxied session-scoped bean. **We can't inject a bean with a lesser scope into a bean with greater scope.**

Attempting to do so, in this case, would trigger an exception with a message containing: *Scope 'session' is not active for the current thread*.

If we're willing to define our controller with session scope, we could avoid specifying a *proxyMode*. This can have disadvantages, especially if the controller is expensive to create because a controller instance would have to be created for each user session.

Note that *TodoList* is available to other components for injection. This may be a benefit or a disadvantage depending on the use case. If making the bean available to the entire application is problematic, the instance can be scoped to the controller instead using *@SessionAttributes* as we'll see in the next example.

5. Using the *@SessionAttributes* Annotation

5.1. Setup

In this setup, we don't define *TodoList* as a Spring-managed *@Bean*. Instead, we **declare it as a *@ModelAttribute* and specify the *@SessionAttributes* annotation to scope it to the session for the controller**.

The first time our controller is accessed, Spring will instantiate an instance and place it in the *Model*. Since we also declare the bean in *@SessionAttributes*, Spring will store the instance.

For a more in-depth discussion of *@ModelAttribute* in Spring, refer to our article on the topic (/spring-mvc-and-the-modelattribute-annotation).

First, we declare our bean by providing a method on the controller and we annotate the method with *@ModelAttribute*:

```
@ModelAttribute("todos")
public TodoList todos() {
    return new TodoList();
}
```

Next, we inform the controller to treat our *TodoList* as session-scoped by using *@SessionAttributes*:

```
@Controller
@RequestMapping("/sessionattributes")
@SessionAttributes("todos")
public class TodoControllerWithSessionAttributes {
    // ... other methods
}
```

Finally, to use the bean within a request, we provide a reference to it in the method signature of a *@RequestMapping*:

```
@GetMapping("/form")
public String showForm(
  Model model,
  @ModelAttribute("todos") TodoList todos) {

    if (!todos.isEmpty()) {
        model.addAttribute("todo", todos.peekLast());
    } else {
        model.addAttribute("todo", new TodoItem());
    }
    return "sessionattributesform";
}
```

In the *@PostMapping* method, we inject *RedirectAttributes* and call *addFlashAttribute* before returning our *RedirectView*. This is an important difference in implementation compared to our first example:

```
@PostMapping("/form")
public RedirectView create(
  @ModelAttribute TodoItem todo,
  @ModelAttribute("todos") TodoList todos,
  RedirectAttributes attributes) {
    todo.setCreateDate(LocalDateTime.now());
    todos.add(todo);
    attributes.addFlashAttribute("todos", todos);
    return new RedirectView("/sessionattributes/todos.html");
}
```

Spring uses a specialized *RedirectAttributes* implementation of *Model* for redirect scenarios to support the encoding of URL parameters. During a redirect, any attributes stored on the *Model* would normally only be available to the framework if they were included in the URL.

By using *addFlashAttribute* we are telling the framework that we want our *TodoList* to survive the redirect without needing to encode it in the URL.

5.2. Unit Testing

The unit testing of the form view controller method is identical to the test we looked at in our first example. The test of the *@PostMapping*, however, is a little different because we need to access the flash attributes in order to verify the behavior:

```
@Test
public void
whenTodoExists_thenSubsequentFormRequestContainsesMostRecentTodo()
throws Exception {
    FlashMap flashMap = mockMvc.perform(post("/sessionattributes/form")
      .param("description", "newtodo"))
      .andExpect(status().is3xxRedirection())
      .andReturn().getFlashMap();

    MvcResult result = mockMvc.perform(get("/sessionattributes/form")
      .sessionAttrs(flashMap))
      .andExpect(status().isOk())
      .andExpect(model().attributeExists("todo"))
      .andReturn();
    TodoItem item = (TodoItem)
result.getModelAndView().getModel().get("todo");

    assertEquals("newtodo", item.getDescription());
}
```

5.3. Discussion

The *@ModelAttribute* and *@SessionAttributes* strategy for storing an attribute in the session is a straightforward solution that **requires no additional context configuration or Spring-managed *@Bean*s**.

Unlike our first example, it's necessary to inject *TodoList* in the *@RequestMapping* methods.

In addition, we must make use of flash attributes for redirect scenarios.

6. Conclusion

In this article, we looked at using scoped proxies and *@SessionAttributes* as 2 strategies for working with session attributes in Spring MVC. Note that in this simple example, any attributes stored in session will only survive for the life of the session.

If we needed to persist attributes between server restarts or session timeouts, we could consider using Spring Session to transparently handle saving the information. Have a look at our article (/spring-session) on Spring Session for more information.

As always, all code used in this article's available over on GitHub (https://github.com/eugenp/tutorials/tree/master/spring-web-modules/spring-mvc-forms-thymeleaf).

Comments are closed on this article!