tapestry-conversations guide

Version status: 0.1.2 release stable, in production
0.1.2 is tested to work with T5.1.0.5 and T5.2.1, T5.2.2 and T5.2.3!
(org.trailsframework:tapestry-conversations:0.0.3 last release to work in 5.0.x Tapestry)
0.1.x interfaces won't change.
0.2.x might introduce new onStartConversation and onContinueConversation events instead of relying on onActivate
Want to see conversations in action? **Check out our live tapestry-conversations example!**

Conversations for Tapestry5!

Have you ever wished to have a scope longer than a request but shorter than session?
Have you ever spent hours optimizing your pages just to avoid the use of sessions?
Have you ever wondered why you have to elaborately describe your page flows in a lengthy XML format, when all you want to do is to make your objects survive through redirect-after-post pattern?
Don't you just wish that your web framework would automatically clean up the session-persisted objects once the user is done using them?
Don't you just hate it when you are browsing for that absolutely cheapest hotel at your favorite hotel search site and you start a second search on a new tab, then come back to the first tab only to see the results from the second search being displayed when you click on the next page link?

If you answered yes to any of the questions above, tapestry-conversations is for you 

Tynamo takes the view that a conversation should happen on a single page (obviously interacting with the same page either with redirect-after-post or ajax). I say Tynamo, but **this module is available for Tapestry 5 with no dependencies** to other Tynamo modules. While it's not as a generic as conversations in JBoss Seam or Spring Web Flow, the conversation-within-page convention makes it far simpler to automatically manage the conversation state and discard the conversation when the page is not used anymore. The thinking is that many short-lived, on-going conversations with a tight memory (session) management should generally be more useful in a web application than complex wizard-type flows. A conversation still needs to be opened manually in the code (typically in your onActivate) because you want (you really do, trust me) the conversation to behave deterministically. If you want your page to support multiple tabs (i.e. have multiple conversations using the same page open at the same time), you need some identifier (typically in the url) to keep the conversations separate. Also if your conversation started automatically, you might easily loose the previous conversation and get a new conversational context without being able to notify the user.

In Tynamo conversations, the conversation identifier is stored either as a cookie or as part of the context. Storing it as part of the context makes it possible to have multiple conversations at the same on separate tabs. Cookies are nice because they are transparent and don't change the url, but if the page url doesn't differ in any other way, it's not possible to have more than one conversation open (per user and per page).

To use the feature, you need to add the following dependency to your pom.xml:

```
<dependency>
  <groupId>org.tynamo</groupId>
  <artifactId>tapestry-conversations</artifactId>
  <version>0.1.1</version>
</dependency>
```

For those who enjoy the thrill of living on the bleeding edge  , automatically deployed, unstable snapshots are hosted at Codehaus' CI repository (http://ci.repository.codehaus.org/org/tynamo/tapestry-conversations/) and manually deployed, sometimes more stable snapshots at snapshot repository (http://snapshots.repository.codehaus.org). If you are going to use SNAPSHOTs, I suggest using the CI ones - the manually deployed ones will always lag behind.

Then, on the page you need to add a org.tynamo.conversations.services.ConversationManager:

```
@Inject
private ConversationManager conversationManager;

// inject componentResources to get the page name
@Inject
private ComponentResources componentResources;
```

Depending on which conversation option you use (cookie / context parameter) you have two patterns to choose from. For a page supporting a single conversation at a time you can do:

```
@Persist("session")
private String conversationId;
```

Then you create the conversation in onActivate:

```
public void onActivate() {
  if (!conversationManager.isActiveConversation(conversationId)) {
    myConversationalProperty = random.nextInt(10);
    // This conversation becomes idle when not used within 60 seconds
    // The last boolean parameter is useCookie, false by default
    conversationId =
conversationManager.createConversation(componentResources.getPageName(), 60, true);
  }
}
```

Note using @Persist("session") makes it clear there can be only conversation at a time. All other persisted properties can be decorated as follows:

```
@Persist("conversation")
private Integer myConversationalProperty;
```

Or, with pages using conversation scope, it's generally easier to use meta annotation @Meta("tapestry.persistence-strategy=conversation") so you don't have to explicitly specify strategy in @Persist annotation. This works well for components with persisted properties as well.

If your page should support multiple conversations, the usage pattern is as follows:

```
private String conversationId;

private Object createConversation() {
  conversationId =
conversationManager.createConversation(componentResources.getPageName(), 60);
  return this;
}

public Object onActivate() {
  if (myConversationalProperty == null) return createConversation();
  else return null;
}

public Object onActivate(String conversationId) {
  if (!conversationManager.isActiveConversation(conversationId) ) return
createConversation();

  if (myConversationalProperty == null) myConversationalProperty = random.nextInt(10);
  this.conversationId = conversationId;
  return null;
}

String onPassivate() {
  return conversationId;
}
```

Notice how we force a redirect in onActivate() (the one without parameters) if the page is not initialized (and the conversation hasn't been started). In onActivate that accepts the conversationId as a parameter, we have to check if it's valid conversation an if not, we again force a redirect. Note that we are not persisting conversationId in this case, but simply relying on passing it along using the activation context. If you choose to keep conversationId as a context parameter, the implementation assumes the conversation id is **the last** parameter of your activation context. Certainly there are other alternatives for creating and using a conversational scope, but the important thing is to keep in mind that you need to manage the conversation identifier to get a hold of the conversation and that when you create a new conversation, a user should be able to return to it even if he refreshes the browser.

StaleStateExceptions?
If you are using Hibernate and you've never seen StaleStateExceptions, you just haven't used it long enough. You'll get a StaleStateException when you are trying to commit a detached entity. Obviously using conversations with a session-per-request pattern will make StaleStateExceptions a much more frequent issue. You could technically carry the whole session in the conversation, but it'll make the conversation heavy. Contrary to what you might have read, using detached objects is just fine. There's way too much misinformation about this subject, but you **can** reattach detached entities to a new session, and the only way to do that is with *session.lock(entity, LockMode.NONE)*. There are some limitations though, and unfortunately, you really have to understand what you are doing, but in principle, that's the way to do it. merge(), get() or load() will **not** reattach, rather, they all return a new (attached) object. See the related stackoverflow question on re-attaching for more info.

Okay, now that you know the principle and options you have for creating conversations, let's focus on how you might want to use conversations in real world. At times it may be useful to start with an empty conversational context, but in practice a much common use case is that you want to initialize a page with some context and then start a conversation in that context, or perhaps create and modify new objects relevant in that context. Let's say you have a Class (Course) that you want to register for. The registration involves filling out several fields and includes semantic validation so you can enter some values, submit them partially and advance to next section (for example you can take the same Course at different times, so you first pick the time you want, etc.) The system accepts Course as the initial context, then creates a Registration object and starts a conversation, letting you modify the same Registration object while making multiple round-trips to the server and back. You can also have several registrations available on different tabs so you can make the schedule work for you. The code could be something like this:

```
@Meta("tapestry.persistence-strategy=conversation")
public class RegisterForCourse {

private String conversationId;

@Persist
@Property
private Registration registration;

@InjectPage
private Index index;

// Handle all context cases in the same onActivate with EventContext
Object onActivate(EventContext context) {
 if (context.getCount() <= 0) return index;
 else if (context.get(String.class, 0).startsWith("registration")) {
  if (conversationManager.isActiveConversation(context.get(String.class, 0))) {
   conversationId = context.get(String.class, 0);
   return true;
  } else return index;
 } else try {
  // Try to coerce the context param into a Course object and
  // re-direct to the conversational context if it succeeds
  return createConversation(context.get(Course.class, 0));
 } catch (Exception e) {
  // Invalid context, return to the home page
  return index;
 }
}

String onPassivate() {
 return conversationId;
}

private Object createConversation(Course course) {
 conversationId = conversationManager.createConversation(
  "registration" + System.currentTimeMillis(), componentResources.getPageName(), 60,
false);
 registration = new Registration(course);
 return this;
}
...
```

Idle conversations are ended automatically when new conversations are started. There's also an optional ConversationModerator component that can be used to end a conversation when user navigates away from the page, to check if a conversation has been idle for too long or to warn the user that the conversation is about to become idle. Any request to the same page (render, component event) will reset the idle timer by default. If you do not want that to happen you can pass in "keepalive=false" as a URL parameter.

Using ConversationModerator component

ConversationModerator is an automatic, invisible ajax component. To use it on a page, add:
<t:conversation.moderator/>

There are several parameters that you can set that change the behavior of the component, the following table should explain them:

name	description	default value
idlecheck	the initial delay after which component checks if the current conversation has become idle. If idlecheck is set, the timing for subsequent checks depends on the keepalive value and the conversation's maxIdleSeconds value	15
warnbefore	the initial delay after which component checks if the current conversation has become idle	15
warnbeforehandler	a string identifier for the Javascript function to be called if warnbefore is set. Of form [object.property.]function. If the identifier specifies context (e.g object.property) **this** is set to that context. If not set, but warnbefore is set, displays an alert box	null
endedhandler	Similar to warnbeforehandler, but called after the conversation has becomed idle.	null
keepalive	Set to make idlecheck to reset the idle counter. Useful when you want to make sure a conversation doesn't become idle until user closes the browser	false

In essence, a ConversationModerator allows you to maintain the memory resources more tightly on the server with small increase in network traffic. You might wonder why you couldn't just use browser's onUnload event to end conversations. Obviously that won't work if users just leave the pages open in their browsers (as people frequently do), but more importantly it will not work either when users refresh the page or when a user invokes a (non-ajax) action on the page, causing redirect-after-post. The few extra requests the component makes for more conservative server session management should be well worth it.

Using ConversationAware (new in 0.1.1)

Sometimes, for example when you are allocating limited resources, you want to make your services aware of conversation start and end events. The classic example for this is buying tickets. A user is interested in buying particular tickets, but he needs some time to reach the buying decision and fill out the payment details. There's a high demand for the tickets so from the business perspective, you only want to reserve the tickets for a single user for a limited time and know as soon as possible if user ends the conversation (willful or not - he could get distracted, his browser could crash in between, etc.). ConversationAware interface allows you to implement support for these scenarios. Your interfaces can implement **ConversationAware** and you can contribute your service as a listener of particular page-specific conversations. For example:

```
 public static void contributeConversationManager(MappedConfiguration<Class,
ConversationAware> configuration,
   TicketMaster ticketMaster) {
  configuration.add(BuyTickets.class, ticketMaster);
 }
```

You can also register/unregister ConversationAware objects as listeners of conversation dynamically using ConversationManager service (which implements a standard listener-notification pattern). Note that you need to specify the page class of which conversations a service is interested in listening (so a particular service doesn't have to listen to all conversations on all pages).

ConversationAware interface specifies two operations:

```
public void onConversationCreated(Conversation conversation);
public void onConversationEnded(Conversation conversation, boolean expired);
```

It's straight-forwarded to use the Conversation object. For example, to release resources reserved temporarily to a particular user (session) you would do something like:

```
public void onConversationEnded(Conversation conversation, boolean expired) {
 freeExistingReservation(conversation.getSessionId());
}
```

For a comprehensive example, see the source code for tynamo-example-conversations that includes an implementation of limited number of comment spots that are allocated to users for a limited time using ConversationAware interface.