

SessionPagePersistence

NOTE: This is outdated information that applies only to Tapestry 4. For current information on page-level persistence, see

<http://tapestry.apache.org/persistent-page-data.html>

Tapestry's session persistence is great, but some times you just want a value to work like a "session page instance variable" - only around while the user is working within a page (calling listener methods on that page, but not rendering any other page). This strategy avoids all the problems of using real instance variables in a page object.

Values are discarded if the last page rendered was not the same as the current page being rendered (see below for another implementation).

You'll need a [PropertyPersistenceStrategy](#), some hivemodule magic, and a [PageEndRenderListener](#). Once it's done, just use `@Persist("session:page")` to take advantage of it.

The strategy:

```
package com.fireapps.tapestry.infrastructure;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;

import org.apache.hivemind.util.Defense;
import org.apache.tapestry.engine.ServiceEncoding;
import org.apache.tapestry.record.PropertyChange;
import org.apache.tapestry.record.PropertyPersistenceStrategy;
import org.apache.tapestry.record.RecordUtils;
import org.apache.tapestry.record.WebSessionAttributeCallback;
import org.apache.tapestry.web.WebRequest;
import org.apache.tapestry.web.WebSession;

/**
 * Derivative of SessionPropertyPersistenceScope
 * - only stores properties while the given page is the most recently rendered page.
 */
public class SessionPagePropertyPersistenceStrategy implements PropertyPersistenceStrategy {

    /**
     * unique persistence name
     */
    public static final String STRATEGY_ID = "session:page";

    /**
     * Session key for the last rendered page name
     */
    public static final String LAST_RENDERED_PAGE_ID = "lastPage";

    // Really, the name of the servlet; used as a prefix on all HttpSessionAttribute keys
    // to keep things straight if multiple Tapestry apps are deployed
    // in the same WAR.

    private String _applicationId;
    private WebRequest _request;

    public void store(String pageName, String idPath, String propertyName, Object newValue) {
        Defense.notNull(pageName, "pageName");
        Defense.notNull(propertyName, "propertyName");

        WebSession session = _request.getSession(true);

        String attributeName = RecordUtils.buildChangeKey(STRATEGY_ID, _applicationId, pageName,
idPath, propertyName);

        session.setAttribute(attributeName, newValue);
    }

    public Collection getStoredChanges(String pageName) {
        Defense.notNull(pageName, "pageName");
```

```

        WebSession session = _request.getSession(false);

        if (session == null) return Collections.EMPTY_LIST;

        /*
         * Single largest difference between this and Session persistence:
         * If the last rendered page is not this page (or is null) then don't return any
         * property values, and set any values found to null.
         */
        String lastPage = (String) session.getAttribute(getLastPageKey(_applicationId));
        if (lastPage == null || ! lastPage.equals(pageName)) {
            // discard values and return empty list
            discardStoredChanges(pageName);
            return Collections.EMPTY_LIST;
        }

        final Collection result = new ArrayList();

        WebSessionAttributeCallback callback = new WebSessionAttributeCallback() {
            @SuppressWarnings("unchecked")
            public void handleAttribute(WebSession ws, String name) {
                PropertyChange change = RecordUtils.buildChange(name, ws.getAttribute(name));
                result.add(change);
            }
        };
        RecordUtils.iterateOverMatchingAttributes(STRATEGY_ID, _applicationId, pageName, session,
callback);
        return result;
    }

    public void discardStoredChanges(String pageName) {
        WebSession session = _request.getSession(false);
        if (session == null) return;
        WebSessionAttributeCallback callback = new WebSessionAttributeCallback() {
            public void handleAttribute(WebSession ws, String name) {
                ws.setAttribute(name, null);
            }
        };
        RecordUtils.iterateOverMatchingAttributes(STRATEGY_ID, _applicationId, pageName, session,
callback);
    }

    public void addParametersForPersistentProperties(ServiceEncoding encoding, boolean post) {
        // nothing to do - we don't use query parameters for sessions
    }

    /**
     * @param applicationName (injected by HiveMind) for uniqueness of session attribute names
     */
    public void setApplicationId(String applicationName) {
        _applicationId = applicationName;
    }

    /**
     * @param request (injected by HiveMind)
     */
    public void setRequest(WebRequest request) {
        _request = request;
    }

    /**
     * @param appId
     * @return application specific session key name for the last rendered page
     * static so the PageEndRenderListener can also use it - there may be a better way for this
     */
    public static String getLastPageKey(String appId) {
        return new StringBuilder().append(STRATEGY_ID).append(",").append(appId).append(",").toString();
    }
}

```

The hivemodule config:

```
<service-point id="SessionPagePropertyPersistenceStrategy" interface="org.apache.tapestry.record.PropertyPersistenceStrategy">
    Stores properties in the session, but only until the next useage of the page.
    Mapped to the name "session:page".
    <invoke-factory>
        <construct class="com.firebaseio.tapestry.infrastructure.SessionPagePropertyPersistenceStrategy">
            <set-object property="request" value="infrastructure:request"/>
            <set-object property="applicationId" value="infrastructure:applicationId"/>
        </construct>
    </invoke-factory>
</service-point>

<contribution configuration-id="tapestry.persist.PersistenceStrategy">
    <strategy name="session:page" object="service:SessionPagePropertyPersistenceStrategy"/>
</contribution>
```

The Page{{ End}}Render`Lister (example shows the definition of an anonymous inner class)

```
new PageEndRenderListener() {
    /**
     * After rendering, store the page name in the session so the "session:page" persistence strategy can
     * use it.
     * @see org.apache.tapestry.event.PageEndRenderListener#pageEndRender(org.apache.tapestry.event.
     PageEvent)
     */
    public void pageEndRender(PageEvent e) {
        final String pageKey = SessionPagePropertyPersistenceStrategy
            .getLastPageKey(e.getRequestCycle().getInfrastructure().getApplicationId());
        e.getRequestCycle()
            .getInfrastructure()
            .getRequest()
            .getSession(true)
            .setAttribute(pageKey, e.getPage().getPageName());
    }
}
```

You need to add this Page{{ End}}Render`Listener to all your pages, unless someone has a neat trick for injecting it that I haven't figured out yet (improvements highly encouraged!).

What we did was add this pageAttached() method to our root page class (extends BasePage, ancestor for all our pages):

```
public void pageAttached(PageEvent event) {
    addPageEndRenderListener([above listener code, to create an anonymous inner class]);
}
```

A completely different implementation for the same purpose (of implementing a session:page strategy) is presented below. The advantage to the solution above is, that it is stand alone and does not rely on the [PageEndRenderListener](#) impl. However, there are some disadvantages to it as well, as it breaks encapsulation of some Tapestry internals (RecordUtils in particular) and might incur performance-penalties when you have lots and lots of properties in your session (because it iterates over *all* session properties for each request).

The strategy:

```
package de.rwth.rz.tapestry.record;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Iterator;

import org.apache.hivemind.util.Defense;
import org.apache.log4j.Logger;
import org.apache.tapestry.IRequestCycle;
import org.apache.tapestry.engine.ServiceEncoding;
```

```

import org.apache.tapestry.record.PropertyChange;
import org.apache.tapestry.record.PropertyPersistenceStrategy;
import org.apache.tapestry.record.RecordUtils;
import org.apache.tapestry.record.SessionPropertyPersistenceStrategy;
import org.apache.tapestry.record.WebSessionAttributeCallback;
import org.apache.tapestry.web.WebRequest;
import org.apache.tapestry.web.WebSession;

/**
 * <p>
 * Strategy similar to client:page but for server properties.
 * </p>
 *
 * <h4> Implementation notes: </h4>
 *
 * <p>
 * Can't extend {@link SessionPropertyPersistenceStrategy} because all methods use STRATEGY_ID which
 * is final in <code>SessionPropertyPersistenceStrategy</code>. So basically this is lot's of
 * duplicated code from said class, with changed strategy id and a one-line change in
 * {@link #getStoredChanges(String)}.
 * </p>
 *
 * <p>
 * Moreover implementation of {@link #evictStaleProperties(String)} is based upon internal knowledge
 * of {@link RecordUtils} workings (yeah, sucks). It works by iterating over all session properties
 * and discarding those properties which have been recorded by this strategy (together with the page
 * name they originated from) but don't belong to the current page.</p>
 *
 * <p>
 * To top it all off, I don't even know whether or not it's correct to hook into
 * <code>getStoredChanges</code> for evicting properties. I don't really know what that method is
 * doing in the first place but I guess it returns those changed properties which should be handled
 * by the page recorder for distribution over a clustered session or something (but that's just a
 * wild guess).
 * </p>
 *
 * <p>
 * The strategy seems to work though.
 * </p>
 *
 * <p>
 * When using this strategy be aware what it means when the user uses the browser back button in
 * order to return from another page to the page before: The page before will have lost all it's
 * page persistent properties, so keep that in mind.
 * </p>
 *
 * @author lehmacher
 */
public class PageSessionPropertyPersistenceStrategy implements PropertyPersistenceStrategy {

    private static final Logger logger = Logger.getLogger(PageSessionPropertyPersistenceStrategy.class);

    /**
     * For page names starting with that prefix no properties are evicted. Basing this on the name
     * is brittle at best, but when I tried retrieving the current page from the request cycle in
     * getStoredChanges method (in order to base that change on an implemented IntermediatePage
     * interface), it did not work (forgot the exact reason why it did not work).
     */
    public static final String SKIP_PREFIX = "intermediate";
    public static final String STRATEGY_ID = "session-page";

    private String _applicationId;
    private WebRequest _request;
    private IRequestCycle _cycle;

    public void store(String pageName, String idPath, String propertyName, Object newValue) {
        Defense.notNull(pageName, "pageName");
        Defense.notNull(propertyName, "propertyName");

        WebSession session = _request.getSession(true);

```

```

        String attributeName = RecordUtils.buildChangeKey(STRATEGY_ID, _applicationId, pageName,
                idPath, propertyName);

        session.setAttribute(attributeName, newValue);
    }

    public Collection getStoredChanges(String pageName) {
        Defense.notNull(pageName, "pageName");

        evictStaleProperties(pageName);

        WebSession session = _request.getSession(false);

        if (session == null)
            return Collections.EMPTY_LIST;

        final Collection result = new ArrayList();

        WebSessionAttributeCallback callback = new WebSessionAttributeCallback() {
            public void handleAttribute(WebSession session, String name) {
                PropertyChange change = RecordUtils.buildChange(name, session.getAttribute
(name));

                result.add(change);
            }
        };

        RecordUtils.iterateOverMatchingAttributes(STRATEGY_ID, _applicationId, pageName, session,
                callback);

        return result;
    }

    public void addParametersForPersistentProperties(ServiceEncoding encoding, boolean post) {
    }

    public void discardStoredChanges(String pageName) {
        WebSession session = _request.getSession(false);

        if (session == null)
            return;

        WebSessionAttributeCallback callback = new WebSessionAttributeCallback() {
            public void handleAttribute(WebSession session, String name) {
                session.setAttribute(name, null);
            }
        };

        RecordUtils.iterateOverMatchingAttributes(STRATEGY_ID, _applicationId, pageName, session,
                callback);
    }

    /**
     * Iterates over all persistent properties and removes those which are page persistent but where
     * current page does not match recorded page.<p/>
     *
     * Implementation breaks encapsulation of RecordUtils.
     */
    protected void evictStaleProperties(String currentPageName) {

        // no session? do nothing
        WebSession session = _request.getSession(false);
        if (session == null)
            return;

        // this is for assuring that properties don't get evicted when a page is retrieved from the
        // request cycle from another page. For example CurrentPage is the current page and it looks
        // up AnotherPage in one of its listener methods. Persistent properties will be restored
        // for AnotherPage but at the same time we don't want the CurrentPage properties to get
    }
}

```

```

// evicted just yet. The checks works on the assumption that when a new page is being surfed
// to, the request cycle will still return null when the persistence strategy is used to
// get stored changes.
if (_cycle.getPage() != null)
    return;

if (currentPageName.startsWith(SKIP_PREFIX) || currentPageName.indexOf="/" + SKIP_PREFIX) != -1)
    return;

// that's how RecordUtils build the prefix string:
// String prefix = strategyId + "," + applicationId + "," + pageName + ",";

// this breaks encapsulation for building the attributeName keys in RecordUtils
String prefix = STRATEGY_ID + "," + _applicationId + ",";

Iterator names = session.getAttributeNames().iterator();
while (names.hasNext()) {

    String name = (String)names.next();

    // if property is not of this strategy or application we skip it
    if (!name.startsWith(prefix))
        continue;

    // means strategy and application match, but page doesn't
    if (!name.startsWith(prefix + currentPageName)) {
        session.setAttribute(name, null);
        logger.debug("evicting from session " + name);
    }
}

public void setApplicationId(String applicationName) {
    _applicationId = applicationName;
}

public void setRequest(WebRequest request) {
    _request = request;
}

public void setRequestCycle(IRequestCycle cycle) {
    _cycle = cycle;
}
}

```

The hivemodule config:

```

<service-point id="PageSessionPropertyPersistenceStrategy"
    interface="org.apache.tapestry.record.PropertyPersistenceStrategy">

    Property persistence Strategy for session properties with page scope, similar to "client:page".

    <invoke-factory>
        <construct class="de.rwth.rz.tapestry.record.PageSessionPropertyPersistenceStrategy">
            <set-object property="applicationId" value="infrastructure:applicationId"/>
            <set-object property="request" value="infrastructure:request"/>
            <set-object property="requestCycle" value="infrastructure:requestCycle"/>
        </construct>
    </invoke-factory>
</service-point>

<contribution configuration-id="tapestry.persist.PersistenceStrategy">
    <strategy name="session:page" object="service:PageSessionPropertyPersistenceStrategy"/>
</contribution>

```