Session

Alias:

User's Environment
Namespace
Threaded-based Singleton
Localized Globals

Motivation:

Military personnel's activities are tracked while they are in a high-security military installation. Their entry and exit are logged. Their badges must be worn at all times to show they are only where they are supposed to be. Guards inside of the base can assume personnel with a badge have been checked thoroughly at the base entrance. Therefore they only have to perform minimal checks before allowing them into a restricted area. Many people are working in a base at the same time. Each security badge uniquely identifies who that person is and what they can do. It also tracks what the carrier of the badge has been doing.

Secure applications need to keep track of global information used throughout the application such as username, roles, and their respective privileges. When an application needs to keep one copy of some information around, it often uses the *Singleton* pattern [GHJV 95]. The *Singleton* is usually stored in a single global location, such as a class variable. Unfortunately, a *Singleton* can be difficult to use when an application is multi-threaded, multi-user, or distributed. In these situations, each thread or each distributed process can be viewed as an independent application, each needing its own private *Singleton*. But when the applications share a common global address space, the single global *Singleton* cannot be shared. A mechanism is needed to allow multiple "Singletons," one for each application.

Problem:

Many objects need access to shared values, but the values are not unique throughout the system.

Forces:

- Referencing global variables can keep code clean and straightforward.
- Each object may only need access to some of the shared values.
- Values that are shared could change over time.
- Multiple applications that run simultaneously might not share the same values.
- Passing many shared objects throughout the application make APIs more complicated.
- While an object may not need certain values, it may later change to need those values.

Solution:

Create a Session object, which holds all of the variables that need to be shared by many objects. Each *Session* object defines a namespace, and each variable in a single *Session* shares the same namespace. The *Session* object is passed around to objects which need any of its values.

Certain user information is used throughout a system. Some of this information is security related, such as the user's role and privileges. A *Session* object is a good way for sharing this global information. This object can be passed around and used as needed.

Depending on the structure of the class hierarchy, an instance variable for the *Session* could be added to a superclass common to every class that needs the *Session*. Many times, especially when extending and building on existing frameworks, the common superclass approach will not work, unless of course you want to extend object which is usually not considered a good design. Thus, usually an instance variable needs to be added to every class that needs access to the *Session*.

All of the objects that share the same Session have a common scope. This scope is like the environments used by a compiler to perform variable lookups. The principle differences are that the

Session's scope was created by the application and that lookups are performed at runtime by the application.

Since many objects hold a reference to the *Session*, it is a great place to put the current *State* [GHJV 95] of the application. The *State* pattern does not have to be implemented inside of the *Session* for general security purposes, however. *Limited View* data and *Roles* can also be cached in a *Session*. It is important to note that the user should not be allowed to access any security data that may be held within a *Session* such as passwords and privileges. It can be a good idea to structure any application with a *Session* object. This object holds onto any shared information that is needed while a user is interacting with the application.

Example:

A *Session* can be used to store many different kinds of information in addition to security data. The Caterpillar/NCSA Financial Model Framework has a FMState class [Yoder]. An FMState object serves as a *Session*. It provides a single location for application components to access a *Limited View* of the data, the current products that can be selected, the user's *Role*, and the state of the system. Most of the classes in the Financial Model keep a reference to an FMState. A true *Singleton* could not be used because a user can open multiple sessions with different selection criteria, each yielding a different *Limited View*.

Figure 4 Shows FMState from the Financial Model. Security info includes username and role. The security info and selection criteria define the limited views. Each ReportView and ReportModel has a reference back to the FMState so it can access other data.

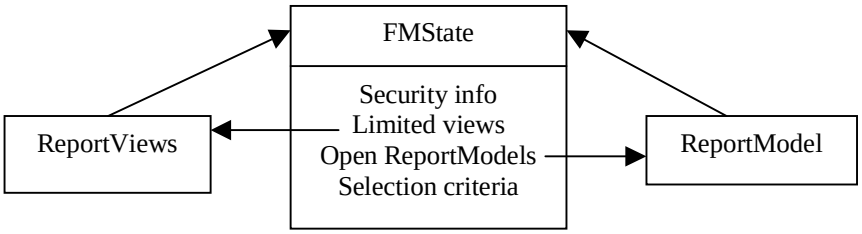

Figure 4 - The Financial Model's FMState

Consequences:

- ✔ The *Session* object provides a common interface for all components to access important variables.
- ✔ Instead of passing many values around the application separately, a single *Session* object can be passed around.
- ✔ Whenever a new shared variable or object is needed, it can be put in the *Session* object and then all components that have access to the object will have access to it.
- ✔ Change propagation is simplified because each object in a thread or process is dependent on only a single, shared *Session* object.
- ✘ While an object may not need a *Session*, it may later create an object that needs the *Session*. When this is the case, the first object must still keep a reference to the *Session* so it can pass it to the new object. Sometimes, it may seem as if every object has a *Session*. The proliferation of *Session* instance variables throughout the design is an unfortunate, but necessary, consequence of the *Session* pattern.

- ✗ Adding *Session* late in the development process can be difficult. Every reference to a *Singleton* must be changed. The authors have experience retrofitting *Session* in place of *Singleton* and can attest that this can very tedious when *Singletons* are spread among several classes. This is also true when trying to consolidate many global variables that were being passed around as parameters into a *Session*.
- ✗ When many values are stored in the *Session*, it will need some organizational structure. While some organization may make it possible to breakdown a *Session* to reduce coupling, splitting the session requires a detailed analysis of which components need which subsets of values.

Related Patterns:

- *Session* is an alternative to a *Singleton* [GHJV 95] in a multi-threaded, multi-user, or distributed environment.
- *Single Access Point* validates a user through *Check Point*. It gets a *Session* in return if the user validation is acceptable.
- A *Session* is a convenient place to implement the *State* pattern because the state is needed throughout the application.
- A Session can keep track of the users *Role* and possibly cache *Limited View* data.
- Lea's *Sessions* [Lea 95] discusses the beginning, middle, and end actions performed during resource management. This paper's *Session* pattern, on the other hand, focuses on separating data that cannot go in a *Singleton* because of a shared environment.
- The Thread Specific Storage Pattern [] allows multiple threads to use one logically global access point to retrieve thread-specific data without incurring locking overhead for each access

Known Use:

- For VisualWorks, the Lens framework for Oracle and GemBuilder for GemStone have OracleSession and GbsSession classes respectively. Each keeps information such as the transaction state and the database connection. The *Sessions* are then referenced by any object within the same database context.
- The Caterpillar/NCSA Financial Model Framework has a FMState class [Yoder]. An FMState object serves as a *Session*, while keeping a *Limited View* of the data, the current product/family selection, and the state of the system. Most of the classes in the Financial Model keep a reference to an FMState.
- The PLoP '98 registration program [Yoder & Manolescu 98] has a *Session* object that keeps track of the user's global information as they are accessing the application.
- Most databases use a *Session* for keeping track of user information.
- VisualWave [OS 95] has a *Session* for its httpd services, which keeps track of any web requests made to it.
- UNIX ftp and telnet services use a *Session* for keeping track of requests and restricting user actions.

Non-Security Known Uses:

- VisualWorks has projects that can be used to separate two or more change sets. While information about window placement is also stored in each project, image code is shared among all of the projects. So, projects could be considered non-secured *Session*.