Java EE Architecture, Part Two

Content

- Requirements on the Business layer
- Framework Independent Patterns
- Transactions
- Frameworks for the Business layer

Requirements on the Business Layer

- Statefullness
  - Should the model retain state, that is should the result of one call still be present in the model when the next call is made?
  - State that is user independent is fine.
  - User dependent state (conversational state) causes problems since the presentation needs to associate different instances of model objects with each user.

Requirements on the Business Layer, Cont'd

- When to synchronize with the database?
  - One alternative is to start every call to the model by loading model data from the database and to end it by storing new or updated data.
  - Another alternative is to cache data in the model.
  - Caching improves performance, but to keep the cache coherent introduces a lot of extra complexity.

Requirements on the Business Layer, Cont'd

- Transactions
  - When should transactions start and stop?
  - What transaction isolation should be used?
  - Since the data handled by the transactions is inside the model, the transactions as well should start and stop inside the model. It would be very high coupling to let presentation manage transactions.

Requirements on the Business Layer, Cont'd

- Error management
  - What exceptions should be reported to presentation?
  - Which component is responsible for logging error messages (preferably the same as for presentation)?
  - How do information about errors arrive to those components (preferably the same way as in presentation)?
  - What information shall be logged?

Requirements on the Business Layer, Cont'd

- Internationalization and localization
  - The model does not show anything to the user but it delivers data and error messages that must be internationalized.

Requirements on the Business Layer, Cont'd

- Interprocess communication
  - Should presentation and business be in the same or separate processes?
  - A reason to use separate processes is that they can run on separate nodes which may give better performance and higher availability.
  - Another reason is that it must be possible to call the model directly from clients.

Requirements on the Business Layer, Cont'd

- Interprocess communication, Cont'd
  - The first reason applies only to extremely big or heavily loaded applications.
  - The second reason can be avoided by putting a web service in front of the model.
  - Because of this and because interprocess communication is slow and a possible source of failure it is almost never a good idea to use different processes for presentation and business.

Requirements on the Business Layer, Cont'd

- Security
  - We do not have to worry about security in the business layer if presentation and business run in the same process and database calls are also made from that process.

Requirements on the Business Layer, Cont'd

- What data shall be used for communication with presentation?
  - Not the business logic object, that would give presentation the possibility to bypass the controller which would lead to extremely high coupling.
  - Not primitive data since that would lead to very long and complicated method signatures.
  - Some kind of data transfer objects must be used.

Framework Independent Patterns, Controller

- A controller is a facade for the model.
- Decouples view and model.
- Its task is to know which methods in the model to call (and in what order) to perform the task the user ordered.
- Its task is also to store user state (conversational state) between user interactions if there is such state.

Framework Independent Patterns, Controller, Cont'd

- The controller does not perform any work, it just delegates to the model.
- The controller must not have any dependencies on the presentation layer.
- It might be a good idea to start development with one controller class. As functionality is added it might become too big (bloated controller). Then it is time to divide it, for example in groups of related functionality.

Framework Independent Patterns, Controller, Cont'd

- The controller must have only one method per system operation.
  - Otherwise presentation will have to know the order to call the controller methods which means it is dependent on the model's internals.
  - Presentation will also become responsible for starting and stopping transactions if it is necessary to make more than one method call per system operation.

Framework Independent Patterns, Data Transfer Object (DTO)

- The problem is how to send data between controller and presentation.
  - Sending primitive data creates very long parameter list in the controller and fills it with get/set methods for all that data.
  - Sending the actual business logic objects couples presentation to business.
  - If operations are not performed in a single call transactions must be handled by presentation.
    - Thus, it is not a good idea to have the view retrieve data by calling a number of getters in the controller.

Framework Independent Patterns, Data Transfer Object (DTO), Cont'd

- The solution is to create data container objects (value object, vo, data transfer object, dto) without any logic.

Framework Independent Patterns, Data Transfer Object (DTO), Cont'd

- This seems like a simple solution, but there is one quite big problem. The application will become flooded with DTOs.
  - For a start there will be the need for one DTO for more or less each entity.
  - Then there will be the need for DTOs that only contain part of an entity's data, or data from many entities.

Framework Independent Patterns, Data Transfer Object (DTO), Cont'd

- There are many (but no perfect) ways to handle this:
  - DTOs are not objects but interfaces with only the get methods in the entities. The actual entities are sent to presentation but presentation only knows about the interface type. This way we do not have to write new objects, but can not have DTOs that combine data from different entities.
  - DTOs are instances of **`java.util.Map`**. A drawback is that presentation and business must agree on key names. These might be constants in enums or interfaces. Another drawback is that maps are not type safe.

Framework Independent Patterns, Data Transfer Object (DTO), Cont'd

- If the data consists of rows from the database that shall be presented in a table in the view, then the DTO might be a **`javax.sql.rowset.CachedRowSet`**, which is like a result set without database connection. This avoids transformation of database tables to DTOs, but might make presentation dependent on the database design. It also makes us use plain JDBC instead of an O/R-mapping framework.
- Let each DTO be a new, unique object. This means lots of code but is easy to understand.

Framework Independent Patterns, Data Transfer Object (DTO), Cont'd

- No matter which solution is used the DTOs should as far as possible be immutable, that is it should not be possible to change their data.
  - Variables and classes are final.
  - Frees us from *a lot* of thread and transaction problems since there is no risk that more than one object updates the same DTO.

Framework Independent Patterns, Data Transfer Object (DTO), Cont'd

- Make a special model package for DTOs.
  - Easy to see that only the DTO model package is imported in presentation.

Framework Independent Problems, Database Synchronization

- Cache data in the model.
  - Might be faster because the number of database calls are reduced.
  - *Far* more complicated since there might be problems to keep the cache up to date and since it most certainly introduces thread problems.
  - Blocks other applications using the same data in the database.

Framework Independent Problems, Database Synchronization, Cont'd

- Read from the database at the beginning of each system operation and write to the database at the end of the same operation.
  - The only realistic choice because of the drawbacks with the previous alternative.
  - Causes lots of calls to the integration layer.
  - Is slower, if that is a problem we must solve performance problems with a good design of the database and the database calls. For example caching can be handled by the database itself or by the O/R-mapping framework.

Transactions

- A transaction is a group of operations that are:
  - Atomic, either all or no of the operations are performed.
  - Consistent, The data is left in a valid state.
  - Isolated, transactions do not affect each other even if they are concurrent.
  - Durable, once a transaction has finished the data is saved, no matter what happens afterwards.
  - These four properties are referred to as ACID.

Transactions, Cont'd

- There are two operations that can end a transaction:
  - Commit, all changes made during the transaction are saved permanently.
  - Rollback, All changes made during the transaction are unmade and the data is left in the same state it had before the transaction started.

Transactions, Cont'd

- Servers are multi threaded.
- Transactions is the mechanism to solve threading problems (for example race conditions).
  - Never set locks or use **`synchronized`** in the business layer.
  - Transactions solve race conditions in the database and entity objects loaded from the database, not for instance variables in non-entity objects in the business layer (like for example the controller).

Transactions, Cont'd

- Transactions must leave objects in a consistent state. If a method fails, for example with an exception, the transaction must be rolled back.
  - The same transaction must be used in all methods involved in the operation so that all entities are left in a consistent state.
  - Everything that was updated during the transaction must be undone. Some state will be rolled back by the transaction manager, exactly what differs between transaction managers.
  - What is not undone by the transaction manager must be undone "manually", in the code.

When do we need transactions?

- Do we need transactions when we read data? When we write data? When we read-update-write data?
  - The answer is that we always need a transaction when we access the database. Otherwise we will access the data without caring about locks and that will cause race conditions.

When do we need transactions?, Cont'd

- However, it is only at read-update-write that we need transactions that span an entire system operation.
- Otherwise we can do with transactions that only live during access to the database.

Non Transactional Resources

- Some resources (for example an ATM) can not participate in a transaction.
- Such resources must always be handled last in transactions because they can not be rolled back.

Always use declarative transactions

- Most frameworks allows to handle transactions either declarative or programmatically.
- Always use declarative transaction management since that hands over control to the framework.
  - It is far to easy to forget transaction management or to start/stop transactions in the wrong place if we code transaction handling manually.
  - Programmatic transaction handling causes low cohesion in the business logic code. Its task is not to handle transactions.

Use Cases Spanning More Than One Transaction

- Suppose a user reads some data, updates it and then saves the updated data.
- With a web based user interface this means one HTTP request to read the data and another, later, request to write the updates.
- With Java EE it is not possible to have transactions live longer than an HTTP request.
  - Even if it had been possible it would not bee a good solution since that long-lived transactions would cause performance problems.

Use Cases Spanning More Than One Transaction, Cont'd

- The scenario in the previous slide introduces a race condition since the following might happen.
  1. User 1 reads some data.
  2. User 2 reads the same data.
  3. User 1 updates the data and saves it.
  4. User 2 updates the data and saves it. The updates made by user 1 are now lost.

Use Cases Spanning More Than One Transaction, Cont'd

- The solution to this problem is to associate a version number with the data. The version number should be stored both on the server and the client.
  1. When the data shall be saved the client sends both data and version number to the server.
  2. Before the data is saved, the server checks if the current version number is equal to the number sent by the client.
  3. If the numbers are equal, the operation succeeds. The data is saved and the version number incremented.
  4. If the numbers are not equal the operation fails and nothing is saved. In this case the user should get information that the used data is stale and perhaps also see both the data used and the current state of the data.

Frameworks for the Business Layer

- Spring (VMware)
- Enterprise JavaBeans, EJB (Oracle, part of Java EE)

Spring, Architecture

- Dependency injection works the same way in business as in presentation.
  - Very similar to CDI.
- Both annotations and xml config files can be used for configuration.
  - Xml files are easier to edit for non-programmers and do not require recompilation.
  - Annotations make the association between the configuration and the configured properties much clearer as they are placed next to each other.

Spring, Transactions

- Enable transactions by adding the line **`<tx:annotation-driven/>`** in the **`applicationContext.xml`** configuration file.
- Put a **`@Transactional`** annotation before each class or method that shall use transactions.
  - It is possible to use the annotation above to specify isolation level, how transactions propagate between method calls, time-out period, if the transaction is read-only and for which exceptions the transaction shall be rolled back.

Spring, Transactions, Cont'd

- Spring's transaction handling does not involve instance variables in non-entity objects. This means that race conditions can occur in for example the controller.
  - The best way to solve this is to avoid writing to such variables.

Spring, Transactions, Cont'd

- Another way is to use stateful controllers, but that introduces the problem to associate different controllers with each user.
- Do not use locks or **`synchronized`** since that may give performance problems.

Spring, Transactions, Cont'd

- Remember that transactions must leave objects in a consistent state. If a method fails, for example with an exception, the transaction must be rolled back.
  - Spring's default behavior is to roll back transactions if unchecked exceptions occur. Checked exceptions must be explicitly specified to cause roll backs.
  - When Spring rolls back a transaction it does not reset instance variables in non-entity objects, e.g. the controller. This *must* be done manually.

Spring, Interprocess Communication

- When using spring the business layer must be in the same process as the presentation layer.
- Thus there can not be any interprocess communication.

Spring, Security

- Since the business layer can only be called from the presentation layer, not directly from a remote client, we do not need to worry about security in the business layer.

EJB, Architecture

- Dependency injection managed by CDI.
- Both annotations and xml config files can be used for configuration.
  - Xml files are easier to edit for non-programmers and do not require recompilation.
  - Annotations make the association between the configuration and the configured properties much clearer as they are placed next to each other.

EJB, Architecture, Cont'd

- Makes difference between EJBs and ordinary objects.
  - Only EJBs are managed and can be configured.

EJB, Architecture, Cont'd

- Each EJB consists of an interface defining the business methods and a class implementing this interface.
  - With EJB, as with Spring, this is not mandatory but an advise.
  - The interface must be annotated **`@Remote`** if it is accessed from another process and **`@Local`** if it is accessed from a client within the same process.
  - The class must be annotated **`@Stateless`** if it has no client specific state and **`@Stateful`** if it has.

EJB, Architecture, Cont'd

- At the first glance EJB might seem quite similar to spring but it is much more powerful and heavyweight.
  - Remote creation and invocation of objects.
  - Asynchronous method calls.
  - Resource management of objects (pooling, storing to disc).
  - Objects can have conversational state.
  - Manages security.

EJB, Transactions

- EJBs are transactional by default.
  - The default is that a transaction starts when a method begins and commits when the method ends.
- Transaction behavior can be modified with the **`@TransactionAttribute`** annotation.
  - The only thing that can be modified is transaction propagation.

EJB, Transactions, Cont'd

- The specification does not say anything about how to specify transaction isolation, timeout period or if a transaction is read-only.

EJB, Transactions, Cont'd

- The EJB container rolls back transactions automatically if exceptions are thrown by the JRE or the EJB container, but not when the application throws exceptions.
  - In that case the application must cause the transaction to roll back by calling **`setRollbackOnly()`**.
  - When EJB rolls back a transaction it does not reset instance variables in EJBs. This *must* be done manually.

EJB, Interprocess Communication

- EJBs can be accessed either locally or remotely.
  - If accessed remotely the interface shall be annotated **`@Remote`**, if accessed locally it shall be annotated **`@Local`**.
  - When accessed locally, parameters are object references (as usual). When accessed remotely the objects themselves are copied and sent to the called method.

EJB, Interprocess Communication, Cont'd

- No matter how they are accessed, EJBs can not be created with the **`new`** operator.
- They must be declared with the **`@EJB`** annotation and will be injected by the servlet container:
  - ```
    @EJB
    MyEjb theEjbInstance;
    ```
  - The above works only in a servlet container, it is a bit more complicated in a standalone client.
    ```

EJB, Interprocess Communication, Cont'd

- For stateless beans the calls to the EJB might be routed to whatever EJB instance in the containers EJB instance pool.
  - The container will only route one call at a time to each instance, so there is no need to worry about thread problems.

EJB, Interprocess Communication, Cont'd

- For statefull beans each call is routed to the same instance.
- Concurrent access to bean instances is forbidden.
  - It is up to the client to avoid calling a bean before the previous call has finished.
  - If clients make concurrent calls to statefull beans the container will either throw an exception or serialize the calls.

EJB, Interprocess Communication, Cont'd

- EJBs can also be called asynchronously by clients using the Java Message Service (JMS) api.
- EJBs can also be used as web services endpoints.

EJB, Security

- The **`@RolesAllowed`** annotation can be used on each EJB method (or class) to specify which roles a user may be in to access the EJB.
  - If there is no such annotation then everyone is allowed to access the EJB.
  - The annotation only works if the EJB is called by a client that can authenticate the users, for example a web container. It does not work with for example clients that are standalone Java programs.

EJB, Security, Cont'd

- If the EJB is accessed locally it is enough to check roles in the web container.
  - It may still be a good idea to use the **`@RolesAllowed`** annotation so that the EJB is not called by mistake from code executed by users not in the correct role.

A Third Option, Our Own Very Lightweight Framework Instead of EJB or Spring

- The only reason to use EJB or Spring is often that we need transaction management.
  - Both of them are quite big and introduces extra complexity, which in this case is unnecessary.

Our Own Very Lightweight Framework, Cont'd

- We could instead write our own very small transaction aware framework. It will consist of just one component, an interceptor that manages transaction boundaries.
  - CDI interceptors are described in lecture 3.
  - There is an outline of a transaction interceptor in the CDI manual, **`http://docs.jboss.org/weld/reference/1.0.0/en-US/html/interceptors.html`**

Evaluation of Our Own Very Lightweight Framework

- Advantages compared to Spring and EJB.
  - Less complexity, easier to start and configure.
  - We have full control over the functionality.
  - Can run on any server with CDI support.
- Disadvantages compared to Spring and EJB.
  - New code means new bugs, which can be very tricky to fix since this is multi-threaded code.
  - We must write all functionality ourselves instead of using existing code.
- Regarding execution speed there is probably not much difference between different frameworks.

A Comparison of Spring and EJB

- Advantages of Spring compared to EJB.
  - Integrates easily with more other frameworks.
  - Has more configurable properties.
  - Covers also the presentation and integration layers, even though it has less functionality and is seldom used for those layers.
  - Can run in the servlet container, more comments on this below.

A Comparison of Spring and EJB, Cont'd

- Advantages of EJB compared to Spring.
  - Contains more functionality for the business layer.
  - Easier to use.
  - Has easy-to-use development environment (Netbeans) and server (GlassFish).

Servers for Spring and EJB

- The Spring container is far more lightweight than the EJB container.
- EJB requires a Java EE application server while Spring does not require any server at all.
  - We still need a servlet container for the presentation layer.

Servers for Spring and EJB, Cont'd

- To step up from a servlet server to a Java EE server has traditionally been very big step in complexity, learning cost and server hosting cost.
- Starting from Java EE 6, there are specifications for *Java EE Web Profile* which is a subset of the Java EE technologies, and includes *EJB Lite* which is a subset of the EJB specification.
  - We can very often do fine with the web profile.

Servers for Spring and EJB, Cont'd

- The Java EE web profile, even though more complex than just the servlet container, reduces the overhead of a Java EE application server a lot.
- Examples of Java EE web profile servers are GlassFish, SIwpas, Resin and JBoss.
- An alternative to a web profile server is to run EJB in a servlet container, using for example OpenEJB from Apache. This way we can use a plain servlet server like Tomcat or Jetty.