OpenSubsystems

Dare to be Open

Architecture

Miroslav Halas

www.opensubsystems.org

Table of Contents

Objectives

OpenSubsystems provides architecture to easily develop business applications. Deliverables of this project should be enough to satisfy requirements of most custom-built business application. This is achieved by integrating and supporting best-of-breed open source and commercial projects and industry standards as well as by original OpenSubsystems contributions.

The goal of OpenSubsystems is to develop and deliver fully functional business components that provide the common functionality required by most custom business applications. The business application is defined, as an application that supports and satisfies needs of a business user. By this definition even though the deliverables of OpenSubsystems are intended to be used by other developers who develop these custom business applications, they are not intended to be tools or application platforms.

OpenSubsystems project therefore does not intend to develop application servers that might be required to run the business application although to support the target environments (especially option 1) it might include portions of infrastructure that would be normally provided by application servers.

Special attention is paid to easy learning experience allowing developers of any level to quickly learn the design and implementation patterns. Anybody with some experiences in Java, databases and web or thick client should be able pickup the OpenSubsystems and implement functional business applications with rich features in matter of hours or days instead of weeks and months.

Target environments for OpenSubsystems based application are:

1. PC running either web based or thick client application used by a single user.
2. Server running either web based or thick client application accessed by a small number of users in the organization usually via company intranet.
3. ASP provider running server or cluster of servers for either web based or thick client application accessed by large number of clients and accessed via Internet.

The first option dictates use of lightweight runtime environment so that it can operate with limited (or at least non-server like) resources. It also dictates use of components, which can be freely distributed and installed (hopefully painlessly) on client's PC. The second option dictates very robust and self-managing architecture, which can function without being constantly monitored by dedicated system administrator. The third option dictates architecture and runtime environment, which can scale to support large loads.

Critical element of OpenSubsystems design and implementation is a requirement that all implementation elements have to be easily testable in an automated fashion. OpenSubsystems provides not only implementation of the business functionality but also extended library of automated tests that can be used by anybody to verify that the functional requirements and well as performance objectives are met.

Building better mouse trap

The business application architecture discussed here follows common model-view-controller design pattern adapted for OpenSubsystems environment. The main objective is to create thin and lightweight structure that clearly separates individual tiers and components of the architecture with the least overhead. At the same time it needs to provide the best performance possible and easy to follow implementation.

Those familiar with J2EE standard and EJB technology, can ask reading next paragraphs, why to don't just use the J2EE/EJBs to implement this architecture and achieve our goals, why to reinvent wheel. The answer is, nothing is being reinvented. OpenSubsystems concepts follow the best practices, which were also adopted by J2EE/EJB world and therefore can be easily mapped to it.

The important fact to realize is that the objectives and desired functionality can be easily implemented even without the EJBs and full J2EE servers so why to pay the price (in performance as well as fairly steep learning curve) for the overhead, which is not necessary. The strategy is to implement this architecture in a lightweight fashion so that all the code, components and subsystems can be used without requiring any heavy server side technology. At the same time OpenSubsystems implementation guarantees that the code can be easily incorporated into applications, which runs on a full-featured J2EE server in a managed environment.

Technology

Even thick client applications these days provide functionality or services requiring the Internet access and often act as servers. It is a reasonable requirement that the platform used to run the OpenSubsystems business applications is a web server/servlet engine. There are plenty of lightweight engines available both as a commercial as well as free or an open source implementations. Web application based on J2EE servlet technology has well defined but at the same time simple lifecycle and it is used as a standard delivery format using which OpenSubsystems applications is delivered. This also makes the OpenSubsystems J2EE compatible. Servlets are used as an entry point for web based functionality but all the business logic is implemented using plain old java objects (POJO), to make it easily testable. Additional compatibility and interoperability with J2EE platform is achieved by using attribute-oriented programming techniques when OpenSubsystems implementation classes can be converted into additional J2EE/EJB components during build time.

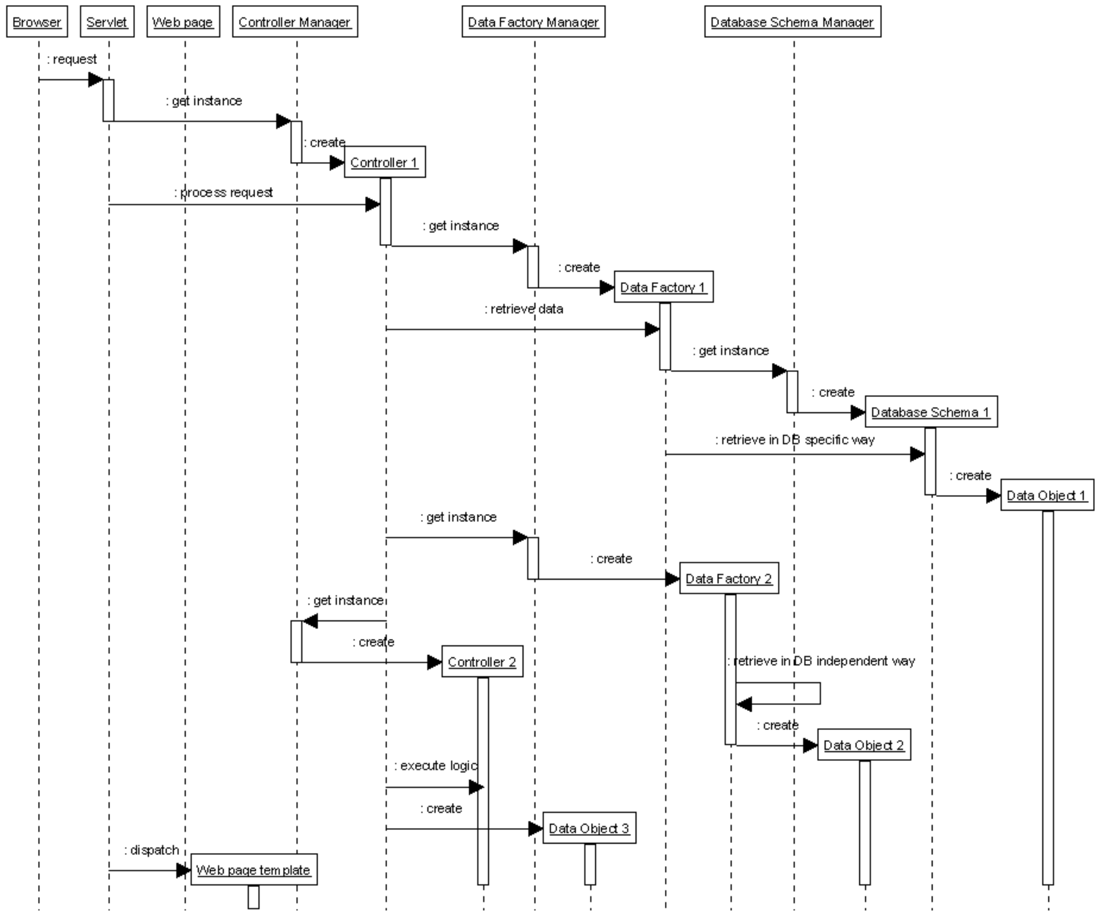
Data flow

The data flow follows what is known as Sun's model 2 architecture. The following example demonstrates how web client would access the functionality of the application. Other clients, such as PDA, thick client or a SOAP client would do it in a very similar fashion possibly skipping the web tier layer and connecting directly to the business logic layer.

The client (in this example browser) generates HTTP request either by typing a URL or by clicking on a hyperlink. Web server receives the request and processes it based on its configuration. Servlet container will be invoked to process the request. Servlet container identifies the context of the web application, invokes any context listeners, filters and at the end servlet, which will handle the request. Business application in this example uses standard web-application format (WAR file or directory structure with WEB-INF and web.xml) to package all context and configure all logic. Web.xml then specifies what servlet will handle what request.

Servlet identifies the request either from the URL or from the passed variable. Every form that gets submitted to the server must contain FORM_NAME hidden variable to make it easier for the server to identify the request being submitted. Servlet determines what business logic it needs to invoke and invokes it without any further delays or processing. The goal is to ensure that the servlet is really there just to parse the request and then invoke execution of the request by the business logic tier. This way any logic, which is invoked by the servlet is reusable by clients other then the browser.

Once the business logic in controller is invoked, it will process the request the same way regardless if the request came from servlet, PDA, SOAP client or thick client. Therefore majority of the code, business logic and functionality will be in controller, because this is the layer, which is reusable by any client. Controller executes any appropriate action. Usually it will get instance of appropriate data factories and coordinates their activities when saving and retrieving data.



Data factory is responsible for creation, modification and retrieval of data from the underlying persistence store. The database is usually the persistence mechanism of choice and there are many ways how to implement database access. In the case of simplest database data factory, it grabs a connection to the database, creates appropriate SQL statements and using JDBC API queries the database.

The results of the data factory calls are returned in the form of data objects. Data object usually encapsulates one row of one table, but it can as well group multiple rows of the same table or all objects involved in parent child relationship. There are no rules or restrictions how the data object should look like and it entirely depends on the application. The goal is to minimize object creation and therefore improve performance.

When the data objects are created, controller can perform any additional transformation of the data. Once all the data processing is done, the result is returned back to the servlet in the web tier in the form of data object. This may be the same data object returned by data factory or it might be newly created data object representing result of processing in the controller. Servlet stores the results in the request object so they can be forwarded for further processing to render the data into HTML.

Servlet based on the current context and request chooses the web page (usually template), which will display the retrieved data. Then it forwards the request, which at this point contains all necessary data objects to the server side presentation tier for rendering.

To summarize the data flow rules:

1. Client (or servlet, which is really just a representation of a client on the server) calls controller (and never calls data factory directly).
2. Controller calls the data factory preferably from it's own subsystem or calls other controllers (if it needs to access data from other subsystems).
3. Data factory calls other data factories preferably from it's own subsystem but never calls controller.

By following these simple rules and structuring the application into these three tiers (client, controller, data factory) the flow of the data and responsibilities of each layer in the application will be clear and straightforward to implement.

Default implementation

Best practices tell us that all tiers are separated from each other by interfaces. Interfaces document the intention and logic, which they implement while leave the implementation upon the requirements of the specific environment. Therefore each controller and each data factory (since they represent boundaries between tiers) provides and should be accessed using its interfaces.

By default the web tier is implemented using servlets and each action is a separate method in a servlet. There is no concept of front controller, because based on the previous experiences it tries to solve an artificial problems. Servlet container already acts as a front controller and dispatches the requests based on the definition in web.xml. There is no need to duplicate this functionality or reinvent different way how to configure it to replace what is already provided by web.xml. Servlet inheritance hierarchy together with filters will take care of the shared services. And finally servlets and web tier exist to just invoke the business logic and should be as thin as possible.

By default, the business logic in controllers is implemented using simple stateless POJOs, which can be easily mapped to stateless session EJBs using attribute-oriented programming techniques. Controllers are implemented as stateless and reentrant objects so that they can be shared between threads of execution and easily replicated in clustered environment.

By default, the persistence in data factories and data objects is implemented using POJOs with direct JDBC calls. It can be easily mapped to entity beans with bean-managed persistence but also it can be completely reimplemented using different persistence mechanism, such as Hibernate. The data factories are expected again to be stateless and reentrant.

One can wonder why JDBC is chosen as a default persistence implementation technology. Based on previous experiences it actually takes about the same time to implement persistence and data retrieval in JDBC as it takes to first figure out what to do in SQL (which any serious query requires) and then convert it to some other persistence mechanism, such as EJB QL. JDBC also gives user much more flexibility in using native database features such as stored procedures, or advanced data accessed techniques such as batched database access, which are often absolutely necessary to achieve desired performance.

Factory pattern

Abstract factory and factory method patterns described by Gof95 are used in all aspects of OpenSubsystems architecture. The idea is that factory is responsible for creation of objects. Since objects are accessed through their interfaces we can change the factory to return different implementation without affecting the rest of the code. Therefore OpenSubsystems provides factories for controllers, data factories and other important implementation dependent elements.

Factories are responsible for instantiation of specific implementation of objects. When a new implementation is provided, factory has to usually change. This is easy if we provide new implementation for a single class but becomes cumbersome when new implementation is provided for a whole set of interfaces such as in case when new persistence mechanism is chosen. OpenSubsystems addresses this issue by implementing concept of factory managers, which are really factories for factories.

The easiest way to demonstrate concept of factory managers is to provide an example. The default implementation of data factories uses JDBC. The convention is that JDBC implementation of interface UserFactory is called UserDatabaseFactory. Default implementation of DataFactoryManager knows how to instantiate JDBC implementation of data factories. The data factory client accesses the UserFactory implementation using code construct (UserFactory)DataFactoryManager.getInstance(UserFactory.class);

If one decides to use different persistence mechanism, for example Hibernate, it will create new implementation of all data factories. By providing new implementation of DataFactoryManager that knows how to instantiate Hibernate implementation of data factories all data factories clients will start using the new persistence layer without a single code change. The new implementation can be based for example on a fact, that the new implementation classes use consistent naming strategy, such as that Hibernate implementation of UserFactory is called UserHibernateFactory.

The same strategy can be used in other places where factory managers are utilized. For example, it is used to provide alternative implementation for business logic layer using stateless session EJBs.

Data access layer

Data access layer provides access to data stored in the underlying persistence store. The persistence store doesn't necessary have to be a database. It can be just a flat file or it can be a data provider sending data through messages. This requires us to create an abstraction of the data access while providing multiple implementations. The most important implementation is a database access layer since most of today's data are stored in the relational databases.

Abstract factory and Data access object patterns are used to model the data provider functionality. OpenSubsystems calls them "data factories". Main responsibilities of factories are creation, modification and retrieval of data. Since each factory can be used by the business logic regardless of the used persistence mechanism, the interface provided by the factory has to be persistence mechanism agnostic (no xxx.sql.xxx imports).

Transfer object pattern (also known as value object pattern) is used to facilitate transfer of data between different layers of application. OpenSubsystems calls them "data objects" and provides DataObject class and few other derived classes, which should be used as base classes for any data object. Main responsibility of data object is to group sets of related attributes so they can be passed around efficiently. Newly created data object classes will most likely contain one attribute for each column in the particular table. The other common design strategy is to create data object for each parent table, which will also contain data for all related child tables.

Database schema

Default persistence store used by OpenSubsystems is a relational database. OpenSubsystems introduces concept of database schema to initialize the database so that it can be used to persist particular data object. Database schema also encapsulates differences in the dialects and persistence strategies for different types of databases.

Database schema might be a new concept for the developers working in environment where DBA takes care of the database. In any other environment, the developer or user of the application is responsible for creation of the database, all the tables, indexes, foreign keys, stored procedures, etc. Database schema is here to makes this process easier.

Database schema is a collection of database tables, indexes and similar database dependent objects, which are related to each other. The purpose of the database schema is to create all the database objects in the database if the database doesn't contain them yet. User can then install OpenSubsystems application and the first time it runs, all what is necessary is created in the database automatically. In a managed environment, DBA would create the database objects ahead of the time and database schema would just detect that nothing needs to be done. This simplifies development as well as deployment.

Each database schema contains version number that should change every time the database schema changes. This should be done by developer or configuration manager when releasing new subsystem. When an OpenSubsystems application containing such modified versioned schema is run for the first time, it can detect what version of the schema the database contains and upgrade it accordingly.

An OpenSubsystems application usually consists of multiple subsystems and therefore multiple database schemas, one for each set of tables, indexes or stored procedures. Each database schema defines set of database schemas it depends on (e.g. via foreign keys or stored procedure calls). On startup, the OpenSubsystems infrastructure verifies the status of the existing schema in the database and creates or updates it if necessary. The OpenSubsystems infrastructure makes sure that all the dependent database schemas are created or upgraded in the database before it tries to create or upgrade the specific database schema.

Unique data identification

Each data object has to have an attribute called ID. A data object has to have the ID attribute even if it has other attributes, which identify it uniquely (such as an email address attribute of data object of type user). Value of the ID attribute must be unique for any particular data object of the same type (e.g. if there is data object of type book, there can be only one book with ID 5, but there can be data object of type dog with ID 5 as well). It is used to uniquely identify that data object in all database and business logic calls. This makes writing the code more straightforward since everybody can rely on the existence and uniqueness of this attribute. Also the database queries such as joins become easier to construct and read.

Value of this attribute is generated when the data object is created and it never changes. For data objects representing row from the database, this attribute should be value of the column populated using unique sequence or ID generator for that database table. This database column would be also primary key for that table. Whenever possible, the value should be generated by the underlying persistence store using sequence, identity column of similar database dependent mechanism instead of application logic.

The value of this attribute should never change. If the value changes it represents a new object. This also applies to scenarios addressing issues such as versioning, each object representing different version should have different ID.

Data partitioning

More often than not, the data needs to be split into some partitions. The partitions can have different purpose. They can be used to assist in load balancing when each partition is stored in a separate persistence store. They can be used to assist in implementation of access control layer where clients can access only the data stored in one of the partitions.

OpenSubsystems establishes notion of a domain representing such partition. Each data object belongs to a domain. Domain can mean anything depending on the business logic of the application. As a consequence of the unique data identification strategy, since the domain is just another data object, domain ID can uniquely identify it.

OpenSubsystems requires that each data object contains domain ID attribute, which identifies the domain the data belongs to. The default implementation allows each data object to belong only to a single domain (even if it is just a default domain) but nothing prevents you to add identifiers for multiple domains into your own data objects.

Data locking strategy

The locking mechanism provided by a database is not going to help resolve problem of concurrent data modification by multiple users. Since the goal of OpenSubsystems is to support many concurrent remote clients using highly scalable sever, at the end of every request the server closes the database connection the client used (or better returns it to the connection pool) and therefore releases any database locks it held. When the client makes the next requests the connection is reestablished or new connection is retrieved from the pool. If it accesses the same data it reacquires the database locks. Of course, any other client could have meanwhile modified the data.

There are two basic locking strategies to resolve this problem. When using pessimistic locking, the data is locked as soon as the user expresses an intention to modify it. This way nobody else can modify the data while the current user is working with it. The data is usually locked for longer time and then at some point when the modifications are finished or when the attempt to modify the data is abandoned, the data is unlocked. This strategy is very effective in a situation when the user wants to be sure, that nobody else can modify the data under the cover. It is especially effective, if conflicts between users occur frequently. It also introduces some additional problems especially in the area of web or remote clients. Since the client is detached from the server, it can be closed, abandoned, disconnected or crash at any time. In this scenario, server needs to detect such situation (usually using timer) and automatically unlock the data. Pessimistic locking therefore requires additional maintenance functionality in the application.

The second strategy is optimistic locking. In this scenario, the application is not trying to establish exclusive lock on the data. It allows user to modify the data but when saving user's modifications it checks, if the data were not modified under the cover by some other user or by the system. If there were some other modifications, user will receive error message about the changed data. The user then has to reconcile the changes. Application can help in the reconciliation e.g. by comparison what changes were made. This strategy is effective if conflicts occur infrequently.

OpenSubsystems uses by default optimistic locking strategy to handle concurrent data modification. The main reasons are the ease of implementation; better scalability if most requests are just read requests and lower administration needs since users will not find themselves locked out of their own data.

Every data object must contain modification date attribute. This attribute must contain the database (or other persistence store) generated timestamp (date + time) when the last modification was made. The timestamp is retrieved from the persistence store together with the data and sent to the client. There it is stored usually using hidden variable on the web page or using internal variable of the thick client. When the modified data are sent to the server, it will contain the original modification date. The original modification date is then used when updating data in the persistence store. Each update contains additional condition statement that can be in SQL expressed as UPDATE ... WHERE .... MODIFICATION_DATE = ? and OpenSubsystems populates it with the modification date sent from client. Persistence store this way modifies the data only if some other user or process did not modify it. System then detects if the update was successful and if not then the user needs to reconcile the conflicts.

Deletion of data objects

It is a common industry practice to distinguish between two types of "deletes" in the persistence store. The first one, "hard delete", actually deletes record from the persistence store. Hard delete functionality is very straightforward and usually causes no problem to implement. When incorrectly or accidentally used, it may of course cause data loss. On the other hand, "soft delete" just marks the record as deleted using some flag but doesn't actually deletes it from the database. Soft delete therefore requires asynchronous cleaning of the persistence store to remove the logically deleted records.

Soft delete is often used in conjunction or instead of data versioning. The developer wants to preserve the original data and therefore she marks the existing record as inactive (soft deleted) and creates a new record. This can often cause unexpected problems, such as a scenario when a developer forgets to exclude such soft deleted records from a query or situation when parent record is logically deleted but the child records are not and therefore still appear as available to application business logic. Another headache may be a situation when the soft deleted records (which because deleted are not visible to users) conflict due to unique constraints on the database with a data that should be newly created. This unfortunately often leads to removing or leaving out the unique constraints from the database design and therefore diminishing the role of a database in the data consistency protection. Soft deleted records can also be a cause of performance problems if they never get purged from the persistence store and therefore skew the database statistics.

Developer should consider if soft delete really makes sense or if what is really desired is ability to version or disable some data. If the requirement is to keep multiple versions of data then constraints on the database will have to take this into account (e.g. name has to be unique between different data items but two versions of the same data item can have the same name). If the requirement is to have ability to disable some data items from being processed in some manner, it is more appropriate to call this functionality enable/disable and implement it as a status flag. This more accurately reflects the purpose of the functionality and doesn't get confused with the real data deletion.

The default mode of OpenSubsystems operation is a hard delete and data is physically deleted from the persistence store when such request is made. If there is a need for soft delete functionality, the approach will be (in terms of relational database used as persistence store) to provide a new table with identical layout as the original table but without any constraints. Every time record is soft deleted it will be instead moved into this new table. This can be implemented by single INSERT ... SELECT SQL statement. The same strategy would be used also for all child records. From the perspective of actively used tables this implementation would simulate the hard delete since the deleted records will no longer exist in the original table therefore avoiding the common problems associated with soft deletes.

Transaction management

Transaction control in JDBC is done per database connection. If the setAutoCommit method is set to false, the transaction is started when the first request is made and the developer can decide about the outcome of transaction by calling commit or rollback on the connection object.

This works just fine for simple database accesses but it is causing problems with more advanced scenarios. First situation where one encounters a problem is an access to two different databases using two different connections. To commit correctly both transactions (since the default JDBC transaction is per connection) one would need two-phase commit protocol, which is not trivial to implement.

The next more common scenario is, if there is a component or subsystem, which is designed and coded to act standalone. Unless it relies on its client to decide transaction boundaries, it needs to issue its commit and rollbacks at some point to work correctly. If this component is reused and in the new usage scenario it is utilized in conjunction with another component then it might be desired for the functionality of both to be performed inside of one transaction. Unless the original component completely relies on its client to coordinate the transactions, the implementation of the original standalone component needs to take into account this new scenario: if there is a transaction then use it, otherwise create a new one. In JDBC transaction world this would require passing and sharing the same connection between multiple components therefore making the client aware of persistence strategy.

The solution to both of these problems is JTA (Java Transaction API) and JTS (Java Transaction Service) specifications, which deal with coordination of multiple parties involved in one transaction. When using JTA based transaction, it is the client of the subsystem (in the web environment the web tier) that decides if transaction is needed or not. If the transaction is needed such as when the interaction with subsystems requires data modification the client will start a transaction and call any business logic accessing any number of databases using any number of components or subsystems. The underlying database layer (not necessarily OpenSubsystems code would automatically check if there is a transaction in progress and associate all necessary resources to this transaction. Once the business functionality is done, the client decides to either commit or rollback the transaction based on the outcome of the subsystem execution. In this scenario the subsystems don't care about the transactions at all, responsibility is completely moved to client of the subsystem. Application or infrastructure needs to provide a transaction manager in order to use JTA/JTS. Transaction manager is responsible for coordination of all resources involved in transaction.

Reality is that full JTA/JTS might be overkill for many applications. In real life, many applications access only single database using only one connection at a time and JDBC based transactions are a sufficient transaction management technique (if one can address propagation of transaction context between multiple subsystems). On the other hand if JTA/JTS is available (such as when running inside of a J2EE server), there is no reason to do not take advantage of it. OpenSubsystems uses concept of database connection factory and transaction factory to provide the best of both world. Subsystems and clients use transaction factory to request transaction to execute business logic in and database connection factory is used to request connections to a databases. Multiple implementations of both are provided to allow use of different connection pools, transaction managers and tight integration with supported middleware. OpenSubsystems also provide an ability to choose either JTA based or JDBC based transaction management completely transparently to all subsystems. Regardless of which one is used, the subsystem clients have the ability to demarcate transaction boundaries if they wish so and individual subsystems have the ability to manage their own transaction if client doesn't provide one.

Web client

Presentation layer is responsible for rendering the data produced by the business logic to the user and coordinating the actions invoked by user. If user uses a thick client application, the application easily keeps track of who is the current user, what area of the application is currently looking at, and what data is the application displaying. When user is using a browser or other type of thin client, the task becomes more complex since the client itself doesn't perform any of these tasks automatically and the application needs to manage the context of execution.

When implementing web clients, web user interface consists of two parts. The first one is the client side presentation layer while the second one is the server side presentation layer. This separation is required because of a simple fact that the user interface is remote from the business logic and it is built on top of stateless HTTP protocol. In between requests the client functions disconnected from the server.

For web applications the client side presentation layer runs inside of a browser and consists of HTML pages that may use CSS to control the layout of components. JavaScript is used to implement business logic since the browser in general only supports JavaScript for client side coding. The server side presentation logic can be based for example on a JSP pages, which run inside of JSP engine bundled with a servlet container or any similar technology such as a XSLT transformation.

Server side presentation layer works very closely with the business logic to respond to users actions. It takes the data objects sent to it as a result of the business logic processing and generates HTML/CSS/JavaScript code for the client side presentation layer. The main design requirement is that the user interface must be easily customizable for different needs and clients. This includes development of a new business logic as well is modification of look and feel of the application for different clients. A person such as web designer who doesn't necessarily know how to code should be able to create or modify the user interface.

OpenSubsystems preferred server side presentation layer technology for the web applications is JSP (Java Server Pages) in combination with custom JSP tags and Java Servlets. On the client side presentation layer it uses JavaScript, CSS, DOM and DHTML to provide rich look and feel of the web applications.

For the client side presentation layer OpenSubsystems introduces inheritance into a web user interface development. Using concept of Tiles OpenSubsystems establishes set of layouts for the development of common types of user interfaces. Developers need to just select the appropriate layout such as dialog, list of items, empty page and fill out the optional parameters. Each web page is inherited from one of the common layouts. Layout can be also inherited from a previously existing layout creating this way dependency hierarchy for functionality as well as look and feel of web pages. Layout acts as a container for a content provided by developer controlling it's position, look and behaviour.

For the server side presentation layer OpenSubsystems utilizes servlet inheritance to control the behaviour of the application. There are two kinds of servlets. There are the ones whose outcome is an HTML page presented in browser. Then there are the ones to implement server related functionality (e.g. upload of files) and don't have any UI. OpenSubsystem provides for both an infrastructure to easily implement Sun's model 2 architecture. Instead of implementing front controller pattern with unnecessary set of extra configuration files, OpenSubsystems uses servlet container as the request dispatcher relying on the default web.xml configuration mechanism. OpenSubsystems utilizes inheritance to provide the common logic such as loading the JSP pages for the UI from the configuration file (either application specific or web.xml) and precaching them and then accessing them later. It takes care of the identifying identity of the user accessing the server and establishing a call context for the business logic execution.