Architekturdokumentation „SAA Online Calendar“

Michael Eckel Christian Krauss

25. Juli 2010

Softwarearchitektur und Anwendungsentwicklung

Prof. Dr. Bodo Alexander Igler

Fachhochschule Gießen-Friedberg
Fachbereich Mathematik, Naturwissenschaften und Informatik

Inhaltsverzeichnis

1 Einführung

Das in dieser Ausarbeitung diskutierte System ist ein Online-Kalender namens „SAA Online Calendar“. Folgende Anforderungen sind an das System gestellt:

- Erweiterbarkeit
- Ereignisorientierung
- Verteiltheit
- GUI

An den entsprechenden Stellen wird deshalb immer wieder Bezug auf diese Anforderungen genommen.

Der Online-Kalender verwendet das Client-Server-Paradigma. Die Funktionalität ist somit aufgeteilt in einen Server und mehrere Clients. Es kann unterschiedliche Arten von Clients geben. In der prototypischen Demo-Applikation stehen zwei alternative Client-Arten zur Verfügung; zum einen eine Web-Oberfläche und zum anderen eine Java-Desktop-Applikation mit Web Service-Schnittstelle. Prinzipiell ist es jedoch möglich, auch andere Arten von Clients zu haben, z. B. einen RMI-Client oder einen in .NET geschriebenen Web Service-Client. Als Programmiersprache und Technologie wurde Java EE verwendet. Auf der Serverseite kommt ein JBoss-Applikationsserver zum Einsatz, der die *Enterprise JavaBeans*-Technologie unterstützt.

Dem Online-Kalender liegt ein Benutzer- und Gruppenkonzept zugrunde. Ein Benutzer kann Mitglied in mehreren Gruppen sein und eine Gruppe kann viele Mitglieder haben. Einem Benutzer ist es erlaubt, Termine anzulegen und diese mit anderen Benutzern und/oder ganzen Gruppen zu teilen. In der prototypischen Demo-Applikation sind die Funktionen auf diese Bestandteile beschränkt, jedoch ist es durch die guten Erweiterungsmöglichkeiten der Anwendung ohne größeren Aufwand möglich, weitere Konzepte, wie das Versenden von Einladungen, einzubauen.

Die Ausarbeitung ist wie folgt gegliedert. Dieses Kapitel gibt eine kurze Einführung in das zu diskutierende System. Kapitel 2 stellt daraufhin die Architektur zuerst auf funktionaler Ebene vor und geht danach auf deren technische Realisierung ein. In Kapitel 3 werden dann ausgewählte „Knackpunkte“ der technischen Realisierung auf Code-Ebene genauer betrachtet. Das abschließende Kapitel 4 diskutiert daraufhin diverse Aspekte der Architektur und geht gezielt auf Vor- und Nachteile im Vergleich zu anderen Architekturen ein.

2 Architektur

In diesem Kapitel wird die Gesamtarchitektur des Systems vorgestellt. Dabei wird in Abschnitt 2.1 zunächst auf die funktionale Struktur der Anwendung eingegangen, bevor dann in den folgenden Abschnitten die technische Umsetzung dieser detaillierter betrachtet wird. Abschnitt 2.2 behandelt die technische Infrastruktur des Servers und Abschnitt 2.3 die des Web Service-Clients.

2.1 Funktionale Struktur der Anwendung

Abbildung 2.1 zeigt ein FMC-Diagramm der funktionalen Struktur des Systems. Es sind zwei Clients zu sehen, ein Web-Client und ein Web Service-Client. Außerdem existiert ein Server. Der Server besitzt eine Kommunikationskomponente, welche aus einem Web-Server und einer Web Service-Schnittstelle besteht. Im Bereich der strukturellen Varianz sind die „Manager"-Komponenten zu sehen, die die Kernfunktionalität der Anwendung bereitstellen. Sowohl der Web-Server als auch die Web Service-Schnittstelle können die Manager-Komponenten benutzen, um Geschäftslogik auszuführen. Üblicherweise verwalten Manager-Komponenten bestimmte Daten. Der *Appointment Manager* kümmert sich um Termine, der *User Manager* um Benutzer und Gruppen und der *Reminder Manager* um Termin-Erinnerungen. Die Daten werden allesamt in einem Speicher (*Storage*) abgelegt.

Es gibt einen Server-Administrator, der dem Server Funktionalität zur Laufzeit hinzufügen kann. Jegliche Funktionalität der Anwendung, auch die Kernfunktionalität, kann also beliebig „hinzugesteckt" und auch wieder entfernt werden. Im Folgenden werden die Manger-Komponenten genauer beschrieben.

Appointment Manager: Ist zuständig für die Verwaltung von Terminen. Dies umschließt das Hinzufügen, Finden, Ändern und Löschen dieser.

User Manager: Ist zuständig für die Verwaltung von Benutzern und Gruppen. Dies umfasst ebenfalls das Hinzufügen, Finden, Ändern und Löschen. Außerdem stehen Funktionen zur Überprüfung der Authentizität und Autorisierung zur Verfügung.

Reminder Manager: Verwaltet Termin-Erinnerungen. Ein Termin kann mit einer Erinnerung versehen werden, welche einen Zeitpunkt, an dem der Benutzer an den Termin erinnert werden will, aufnimmt. Die Funktionalität des *Reminder Managers* umfasst auch die Operationen zum Hinzufügen, Finden, Ändern und Löschen von Termin-Erinnerungen.

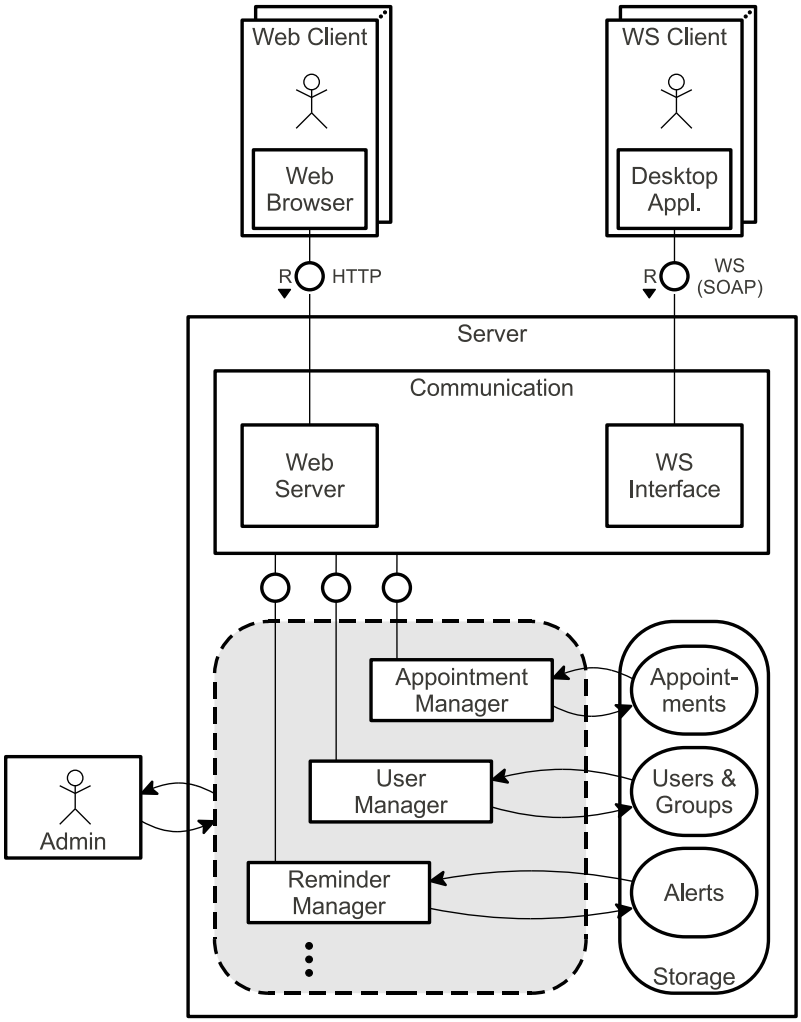


Abbildung 2.1: Funktionale Struktur des Systems

2.2 Technische Infrastruktur des Servers

Die technische Infrastruktur der Anwendung ist in Abbildung 2.2 zu sehen. Es geht darum, die funktionale Struktur auf die technische abzubilden bzw. sie in diese zu transformieren. Dabei sind an einigen Stellen auch nicht-hierarchische Transformationen nötig.

Die gesamte Server-seitige Anwendung läuft im Applikationsserver *JBoss* ab. Dieser bietet einen EJB-Container zur Verwaltung von *Enterprise JavaBeans* (EJBs und einen Web-Server (*Apache Tomcat*) zur Verwaltung von Web-Anwendungen. Zum Auffinden von EJBs dient das *Java Naming and Directory Interface*, kurz JNDI. *Enterprise JavaBeans* liegt die Variante *Asymmetric Extension Interface* des *Extension Interface*-Entwurfsmusters zugrunde. In dieser Variante existiert ein zentraler Dienst, nämlich JNDI, zum Auffinden von „Komponenten“; im Gegensatz zum Original, wo über jede Komponente nach anderen gesucht werden kann.

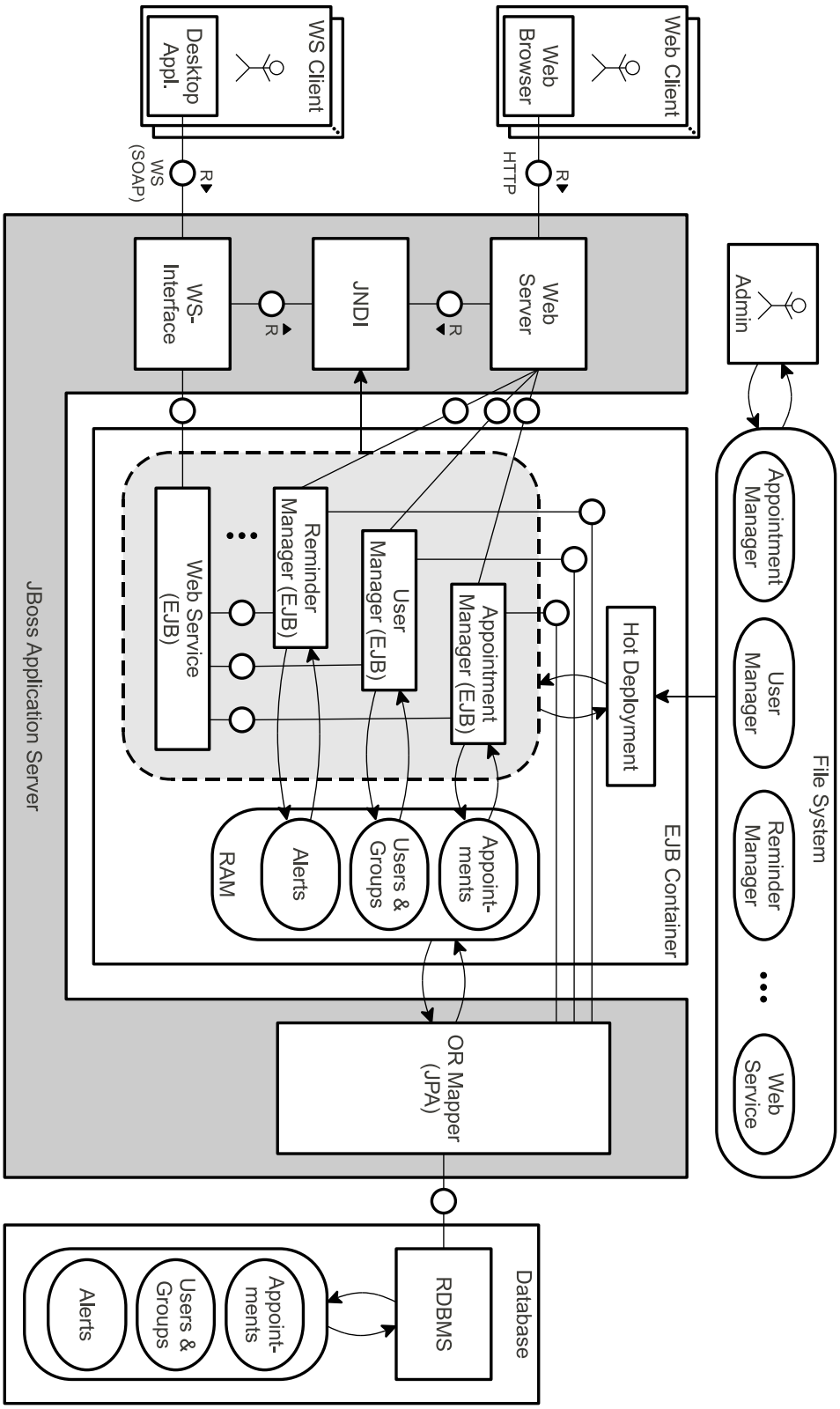


Abbildung 2.2: Technische Infrastruktur des Servers

Die in der funktionalen Struktur zu sehenden Manager-Komponenten werden in der hier vorgestellten technischen Infrastruktur auf jeweils eine EJB gemappt. Im Folgenden werden diverse Aspekte der technischen Infrastruktur näher erläutert.

2.2.1 Nutzung der Serverfunktionalität durch den Web-Client

Der Web-Client nutzt den Web-Browser, um mit dem Web-Server im JBoss-Applikationsserver zu kommunizieren. Der Web-Server seinerseits nutzt die entsprechende Geschäftslogik mit Hilfe der EJBs, welche er über JNDI findet.

2.2.2 Nutzung der Serverfunktionalität durch den Web Service-Client

Der Web Service-Client nutzt die Web Service-fähige Desktop-Anwendung, um mit der Web Service-Schnittstelle des JBoss-Applikationsservers zu kommunizieren. Die Web Service-Schnittstelle nutzt JNDI, um die entsprechende *WebService*-EJB zu finden, mit der sie die Geschäftslogik abwickeln kann. Die *WebService*-EJB fungiert dabei lediglich als Adapter und leitet die Aufrufe an die entsprechenden EJBs durch.

2.2.3 Hot Deployment

Der EJB-Container unterstützt sogenanntes *Hot Deployment*, was bedeutet, dass Funktionalität in Form von EJBs während der Laufzeit hinzugefügt und auch wieder entfernt werden kann. In Abbildung 2.2 ist oben der Administrator zu sehen, welcher EJB-Dateien in den entsprechenden Ordner im Dateisystem legen kann, der von der *Hot Deployment*-Komponente des EJB-Containers auf Änderungen überwacht wird. Stellt die *Hot Deployment*-Komponente das Hinzufügen einer EJB fest, kümmert sie sich um das Laden und Instanziieren der EJB im EJB-Container und das Registrieren beim JNDI. Im Falle des Entfernens einer EJB-Datei durch den Administrator, stellt die *Hot Deployment*-Komponente dies auch fest und veranlasst entsprechend das Deregistrieren und Entladen der EJB.

2.2.4 Persistierung

Als Art der Daten-Persistierung wird objekt-relationales Mapping (ORM) mit Hilfe der *Java Persistence API* (JPA) verwendet. Als konkrete Implementierung kommt *Hibernate* zum Einsatz, welches vom JBoss-Applikationsserver als Standard-Persistenz-Framework mitgeliefert wird. Das dahinter liegende Datenbanksystem der Demo-Applikation ist *PostgreSQL*, welches aber durch jedes andere relationale DBMS ersetzt werden kann. Deshalb ist dieses in Abbildung 2.2 einfach als „RDBMS“ bezeichnet.

Die von den EJBs zu persistierenden Daten bzw. Objekte (vgl. Abschnitt 3.1) werden in den Arbeitsspeicher geschrieben und der OR-Mapper wird angewiesen diese zu persistieren. Die zu

persistierenden Objekte sind allesamt leichtgewichtige POJOs. Der Programmierer muss sich nicht selbst um die Persistierung kümmern, weil sie vom Container bereitgestellt wird (*Container Managed Persistence*). Da die POJOs somit als passive Komponenten anzusehen sind, werden sie in Abbildung 2.2 als Speicher dargestellt.

2.3 Technische Infrastruktur des Web Service-Clients

Abbildung 2.3 zeigt ein FMC-Diagramm mit dem Aufbau des Web Service-Clients (im Folgenden auch einfach WS-Client). Der Benutzer interagiert mit der *View*, welche die GUI darstellt. Die Kommunikation mit dem Server findet über die *Communication*-Komponente statt. Den Kern des WS-Clients bildet die *Core*-Komponente, welche die Kommunikation aller Komponenten untereinander regelt. Das *Plugin Management* ist verantwortlich für das Laden und Entladen der Plugins zur Laufzeit (für Details siehe Abschnitt 3.4). Der Benutzer kann Im- und Export-Plugins in einen bestimmten Ordner im Dateisystem legen und auch wieder entfernen. Das *Plugin Management* erkennt dies und lädt bzw. entlädt das jeweilige Plugin. Alle Plugins können über die *Core*-Komponente angesprochen werden.

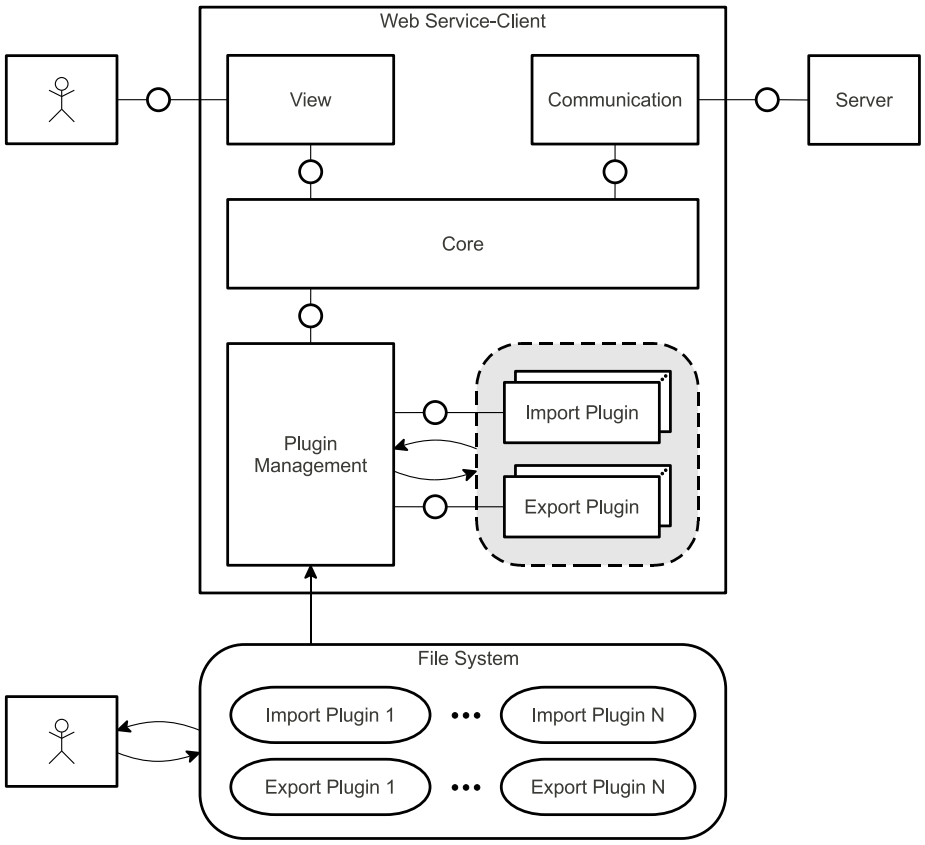


Abbildung 2.3: Technische Infrastruktur des Web Service-Clients

2.4 Verteilung der Infrastrukturkomponenten

Da die Verteiltheit eine der Hauptanforderungen an das System darstellt, wird in diesem Abschnitt kurz beschrieben, wie die Komponenten des „SAA Online Calendar“ verteilt werden. Die Verteilung erfolgt gemäß der 4-Schichten-Architektur, welche für Daten-zentrierte Anwendungen wie den hier zu entwickelnden Online-Kalender sehr gut geeignet ist. Im Folgenden werden die Schichten anhand Abbildung 2.2 erläutert.

Schicht 1 (Präsentation): Die Präsentationsschicht stellen die Clients dar, in Abbildung 2.2 ist dies zum einen der *Web Client* und zum anderen der *WS Client*.

Schicht 2 (Web Server): Obwohl der Web-Server im Kontext des JBoss-Applikationsservers läuft, nimmt dieser hier eine Sonderstellung ein. Er ist verantwortlich für die Interaktion zwischen Geschäftslogik (siehe nächste Schicht) und dem Web-Client (Präsentationsschicht). Er ist also als Controller anzusehen.

Schicht 3 (Geschäftslogik): Diese Schicht wird repräsentiert durch den JBoss-Applikationsserver, ausgenommen dem *Web Server*. Dazu gehören speziell alle EJBs, die die anwendungsspezifische Logik darstellen, aber auch alle Infrastrukturkomponenten, welche als Rahmenwerk für die Geschäftslogik dienen.

Schicht 4 (Persistenz): Die Persistenzschicht wird durch die Datenbank repräsentiert, in der die Daten physikalisch dauerhaft gespeichert werden.

3 Code-Ebene

In diesem Kapitel werden ausgewählte Bestandteile diverser Komponenten des Gesamtsystems detaillierter betrachtet. Die ausgewählten Bestandteile wurden von den Autoren als „besonders knifflig“ oder einfach „interessant“ eingestuft.

3.1 Grundlegende Entitäten und Persistierung

Zur Persistierung der Entitäten wird, wie oben bereits erwähnt, die *Java Persistence API* verwendet. Im UML-Klassendiagramm in Abbildung 3.1 sind die grundlegenden Enitäten des Systems zu sehen. Die oberen beiden Klassen, `User` und `Group`, werden vom User-Manager verwaltet, die Klasse `Appointment` vom Appointment-Manager. Ein Objekt vom Typ `Appointment` repräsentiert dabei einen Einzeltermin (also ohne Wiederholung). Die drei unteren Klassen, `WeeklyAppointment`, `MonthlyAppointment` und `YearlyAppointment`, sind in der prototypischen Demo-Applikation noch nicht realisiert, werden hier jedoch als mögliche Erweiterung für sich wiederholende Termine dargestellt.

Zwischen den Klassen `User` („Benutzer“) und `Group` („Gruppe“) besteht eine *many-to-many*-Beziehung. Ein Benutzer besitzt eine Liste der Gruppen, in denen er sich befindet. Eine Gruppe besitzt eine Liste der Benutzer, die Mitglied in der Gruppe sind. Zwischen den Klassen `User` und `Appointment` („Termin“) bestehen zwei Beziehungen. Erstere ist eine *one-to-many*-Beziehung aus Sicht des Benutzers. Einem Benutzer können mehrere Termine gehören und ein Termin gehört genau einem Benutzer. Die zweite Beziehung ist wieder eine *many-to-many*-Beziehung. Der Benutzer besitzt eine Liste aller Termine, auf die er lesenden Zugriff hat, also auch Gruppentermine oder ausgewählte Termine, die von anderen Benutzern für diesen Benutzer freigeschaltet wurden. Ein Termin enthält eine Liste aller Benutzer, denen erlaubt ist, den Termin zu sehen. Zwischen Gruppen und Terminen existiert eine äquivalente Beziehung, mit dem Unterschied, dass es sich hier um Gruppentermine und nicht um Benutzertermine handelt.

Die Klasse `Appointment` ist mittels JPA-Annotationen so annotiert, dass sie die Basisklasse für eine Vererbungshierachie darstellt. Die Vererbungshierarchie wird mit der Methode `JOINED` in das relationale DBMS abgebildet, was bedeutet, dass genau eine Tabelle pro konkreter Klasse angelegt wird und ein Diskriminator zur Typ-Unterscheidung eingesetzt wird. An dieser Stelle kann das System um zusätzliche Terminarten erweitert werden. Im o. g. Klassendiagramm stellen die unteren drei Klassen, wie bereits erwähnt, mögliche Erweiterungen dar. Für jede zusätzliche Termin-Art wird eine Unterklasse von `Appointment` angelegt und zusätzlich eine neue EJB, die diese Art von Terminen verwaltet. Die gemeinsame Basis ist immer `Appointment`, d. h. alle Termine sind mindestens auf `Appointment` abbildbar und bestehende Funktionalität wird nicht beeinträchtigt. Die neue EJB ist gemäß des *Extension Interface*-Entwurfsmusters

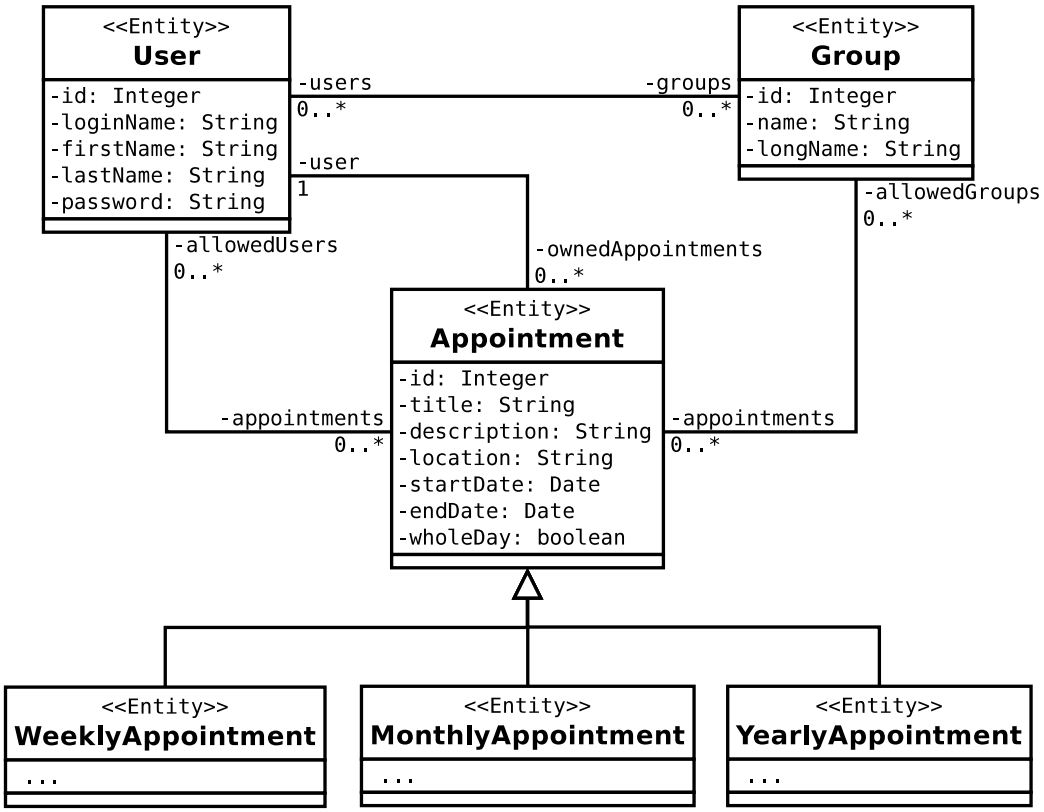


Abbildung 3.1: UML-Klassendiagramm der grundlegenden Entities

ein neues *Extension Interface* und Clients, die dieses unterstützen, können die erweiterte Funktionalität nutzen. Kurz gesagt: Erweiterungen sind möglich, ohne bestehende Funktionalität zu beeinträchtigen; ganz im Sinne des Entwurfsmusters *Extension Interface*.

3.2 Anfragebearbeitung im Web-Server

Im Folgenden wird beschrieben, wie der Web-Server Anfragen des Web-Clients intern verarbeitet. Dabei kommt das Entwurfsmuster *Front Controller* zum Einsatz. Abbildung 3.2 zeigt ein FMC-Diagramm, das die Funktionsweise veranschaulichen soll.

Der Client sendet über den Web-Browser eine Anfrage an den Web-Server. Im Web-Server wird die Anfrage von einem Filter in der *Filter Chain* entgegen genommen. In diesem Fall ist dies der Logging-Filter, der die Aufgabe hat, die Bearbeitungszeit der Anfrage für Statistikzwecke mit zu loggen. Der nächste Filter in der Kette ist der Login-Filter. Dieser stellt sicher, dass der Benutzer eingeloggt ist. Ist dies nicht der Fall, leitet der Filter die Anfrage an die Login-Seite (*Login Page*) weiter, welche durch die Filter zurück an den Client gesendet wird. Der Client kann nun das Login-Formular ausfüllen und zurück an den Web-Server senden. Mit abschicken der Login-Seite wird dem Web-Server eine *Action* übermittelt, in diesem Fall die Login-Action. Die Anfrage geht durch den Logging-Filter an den Login-Filter. Dieser erkennt nun, dass die

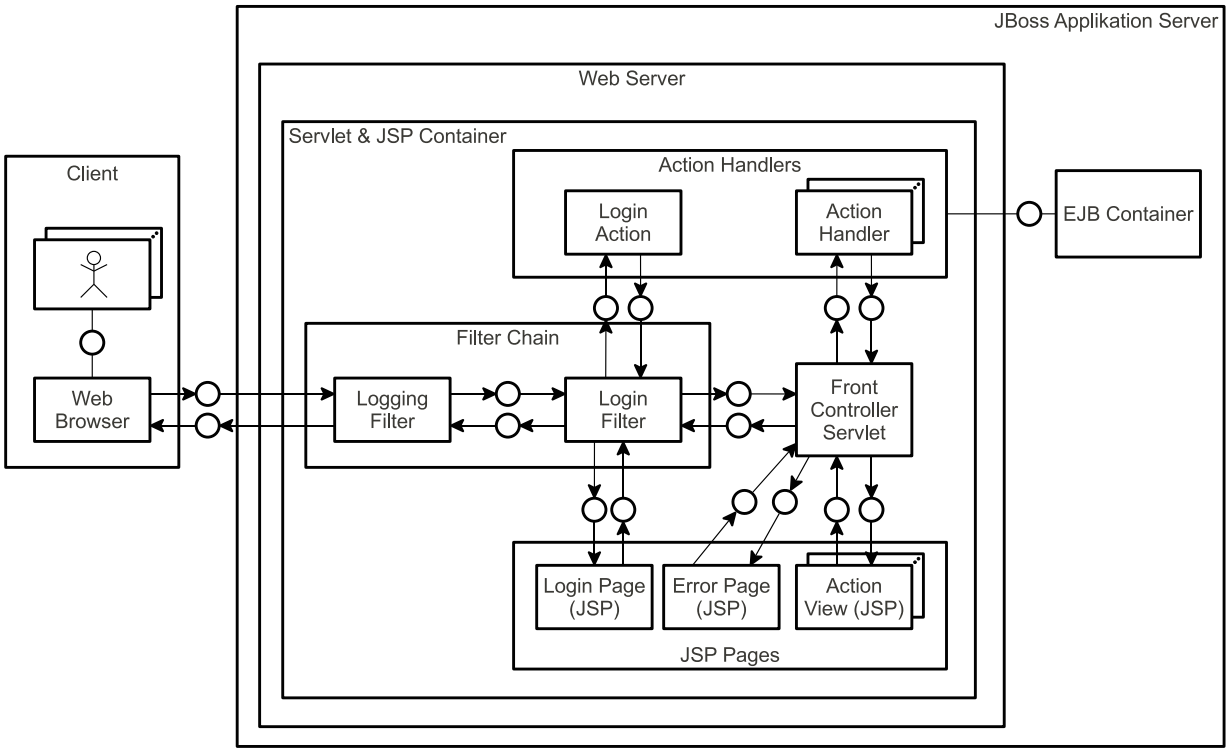


Abbildung 3.2: „Front Controller“-Entwurfsmuster im Web-Server

Login-Action angefragt wurde und prüft mittels der *Login Action* ob die Anmeldedaten korrekt sind. Die *Login Action* nutzt dabei die Funktionalität des *User Managers* im EJB-Container des JBoss. Ist der Login korrekt, wird die Anfrage zum Front-Controller durchgereicht, andernfalls bekommt der Client wieder die Login-Seite präsentiert.

Der Front-Controller ist das Kernstück der Web-Anwendung. Er arbeitet als Dispatcher und Router. Anhand des Namens der Action findet er per Reflection den zugehörigen Action-Handler, welcher durch eine einfache Java-Klasse realisiert ist. Jeder Action-Handler implementiert das Interface `Action`, welches die Methode `execute` deklariert. Hat der Front-Controller keine entsprechende Klasse gefunden, leitet er die Anfrage an die Fehler-Seite (*Error Page*) weiter und übergibt dieser eine entsprechende Fehlermeldung. Die Antwort geht durch den Front-Controller und die Filter zurück an den Client.

Hat der Front-Controller einen entsprechenden Action-Handler gefunden, sprich eine Java-Klasse, erstellt er eine Instanz dieser und ruft die Methode `execute` auf, die ja jeder Action-Handler implementieren muss. Ein Action-Handler kann zum Erfüllen seiner Aufgabe die EJB-Funktionalität nutzen und hat die Möglichkeit, Ergebnisse – falls vorhanden – in den Anfrage-Kontext zu legen. Nach Aufruf des Action-Handlers leitet der Front-Controller die Anfrage an die entsprechende Action-View weiter, welche durch eine Java Server Page repräsentiert wird. Gefunden wird diese wieder anhand des Action-Namens. Sie kann die Ergebnisse des Action-Handlers aus dem Anfrage-Kontext holen und entsprechend als HTML-Seite darstellen. Die Antwort, also die HTML-Seite, geht nach Fertigstellung durch den Front-Controller und die Filter zurück zum Client.

An dieser Stelle sollen noch zwei Dinge noch mal hervorgehoben werden. Zum Einen muss ein Action-Handler immer zum jeweiligen Action-View passen und umgekehrt, d. h. Ergebnisse, die der Action-Handler erzeugt, müssen vom Action-View verstanden werden. Action-Handler und Action-View sind also stark voneinander abhängig. Die zweite Sache betrifft den Front-Controller und die Filter. Der Front-Controller kann sich zu jeder Zeit sicher sein, dass, wenn er aufgerufen wird, der Login bereits geschehen ist. Er braucht sich nicht weiter um die Authentifizierung zu kümmern. Die Nachbedingung des Login-Filters ist also, dass der Benutzer authentifiziert ist.

3.3 Ereignisbasierung des Web-Clients

Da das HTTP-Protokoll bekanntlich zustandslos ist und der Initiator einer Kommunikation immer der Web-Client ist, muss ein Weg gefunden werden, wie der Web-Server dem Web-Client Nachrichten senden kann. Dieses Manko wird mittels AJAX behoben. Dabei läuft in der HTML-Seite JavaScript-Code, der – transparent für den Client – in regelmäßigen Abständen (z. B. 10 Sekunden) beim Web-Server anfragt, ob es Aktualisierungen gibt. Ist dies der Fall, antwortet der Server mit genau diesen Aktualisierungen und der JavaScript-Code kann die HTML-Seite entsprechend modifizieren. AJAX ist also ein Workaround, um den fehlenden Kommunikationskanal vom Web-Server zum Web-Client zu emulieren.

Server-seitig existieren spezielle Behandlungsroutinen für AJAX-Anfragen, die ebenfalls dem in 3.2 vorgestellten *Front Controller*-Entwurfsmuster entsprechen. Um sicherzustellen, dass nur authentifizierte Benutzer AJAX-Anfragen ausführen können, wurde das Konzept eines Security-Tokens eingeführt. Dieses Security-Token wird bei erstmaligen Login in das System generiert und assoziiert mit dem Benutzer im Applikationkontext gespeichert. Das Security-Token, muss bei jeder AJAX-Anfrage vom Client mitgesendet werden.

3.4 Das Plugin-System im Web Service-Client

Beim Plugin-System im Web Service-Client handelt es sich um ein selbst implementiertes. Ein Plugin wird dabei von einer Java-Klasse repräsentiert. Der Knackpunkt an dieser Stelle ist das Laden und Entladen von Klassen zur Laufzeit. Abbildung 3.3(a) zeigt ein UML-Objektdiagramm aller involvierten Komponenten. Die Funktionsweise soll im Folgenden näher erläutert werden, beginnend mit einer Beschreibung der Komponenten.

Client: nutzt das Plugin-System durch verwenden des *DeployManagers*.

DeployManager: verwaltet eine Liste mit genau einer Instanz jedes Plugins und stößt Lade- und Entladevorgänge von Klassen beim *HotDeployLoader* an.

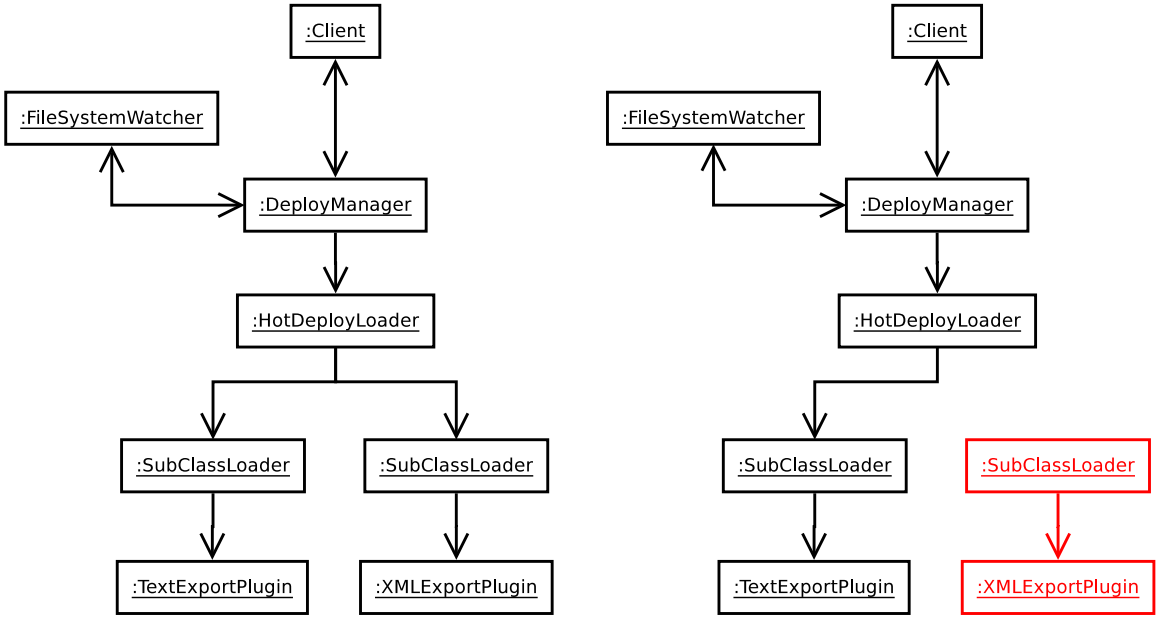
FileSystemWatcher: überwacht das Plugin-Verzeichnis im Dateisystem und meldet Änderungen an den *DeployManager*.

HotDeployLoader: verwaltet das Laden und Entladen von Klassen. Er delegiert das Laden von Klassen an je einen *SubClassLoader*.

SubClassLoader: lädt eine Klasse (und ihre abhängigen Klassen) wirklich.

Laden eines Plugins

Der *FileSystemWatcher* stellt fest, dass ein Plugin (Java-Klasse) im überwachten Verzeichnis hinzugefügt wurde. Diese Änderung meldet er dem *DeployManager* und übermittelt diesem auch den entsprechenden Dateinamen. Der *DeployManager* seinerseits veranlasst den *HotDeployLoader*, die Klasse zu laden. Der *HotDeployLoader* erstellt einen neuen *SubClassLoader* und delegiert den Klassenladevorgang an diesen. Außerdem führt er eine Liste über all seine *SubClassLoader*. Der Rückgabe-Wert des *SubClassLoaders* ist ein Klassenobjekt (`Class<?>`) des Plugins, das vom *SubClassLoader* über den *HotDeployLoader* zum *DeployManager* durchgereicht wird. Der *DeployManager* erstellt nun eine Instanz dieser Klasse und legt sie in seiner Liste vorhandener Plugins ab.



(a) Komponenten des Plugin-Systems (b) Nach dem Entfernen eines Plugins

Abbildung 3.3: Laden und Entladen von Plugins im Web Service-Client

Entladen eines Plugins

Da es in Java nicht möglich ist, Klassen explizit zu entladen, muss ein kleiner Trick angewendet werden, um dies trotzdem (implizit) zu erreichen. Eine Klasse wird vom *Garbage Collector* nur entladen, wenn keine Referenz mehr auf das Klassenobjekt selbst oder eine Instanz der Klasse existiert. Deshalb wurde pro geladener Plugin-Klasse ein eigener Klassenlader (*SubClassLoader*) erstellt.

Wenn der *FileSystemWatcher* nun feststellt, dass ein Plugin entfernt wurde, benachrichtigt er den *DeployManager*, welcher nun die Plugin-Instanz aus seiner Liste entfernt und den *HotDeployLoader* anweist, die Klasse zu „entladen“. Der *HotDeployLoader* entfernt nun denjenigen *SubClassLoader* aus seiner Liste, der die entsprechende Plugin-Klasse geladen hat. Somit existiert keine Refrenz mehr auf diesen. Abbildung 3.3(b) zeigt die „gekappte“ Verbindung und den nicht mehr referenzierten *SubClassLoader* und die Plugin-Klasse. Der *Garbage Collector* kann diese nun beide entladen.

Problem: An dieser Stelle kann es vorkommen, dass im Client-Code noch Referenzen auf Instanzen von Plugins existieren und der *Garbage Collector* somit die Plugin-Klassen und alles damit Zusammenhängende nicht entfernen kann.

Lösung: Entweder man achtet im Client-Code darauf, dass nirgendwo Referenzen auf Plugins zwischengespeichert werden oder man sorgt dafür, dass der Client-Code nie in Kontakt mit direkten Referenzen auf Plugins kommt. Ersteres ist nicht zu empfehlen, da jeder Programmierer, der das Plugin-System verwendet, in seinem Code explizit darauf Rücksicht nehmen muss. Zweiteres ist eher zu empfehlen und kann z. B. durch Einsatz des *Proxy*-Entwurfsmusters gelöst werden. Dabei arbeitet der Verwender anstatt auf dem Original auf einem Objekt mit gleicher Schnittstelle (dem „Proxy“), das die Aufrufe intern an das Original weiterleitet. Somit ist zu jedem Zeitpunkt sichergestellt, dass der Client nur Referenzen auf die *Proxy*-Objekte halten kann und niemals auf das Original.

4 Diskussion der Architektur

[Dieser Teil soll von von Herrn Krauss nachgereicht werden]