1. Gedanken und Überlegungen zu der Importerfunktionalität

1.1 Designentscheidung

Die Aufgabenstellung umfasst die Anforderung an eine Schnittstelle um Aktivitäten aus Fremdsystemen zu importieren und nutzbar zu machen. Da es keine festen Angaben gab, wie das Format auszusehen hat, haben wir uns für das XML Format entschieden. Die Vorteile von XMLs liegen auf der Hand.

* Standardisierung
* Offenheit
* Sehr stark verbreitet
* Sehr gute Erweiterbarkeit der Attribute (Flexibilität)
* Einfachheit

Durch die Baumstruktur (DOM) lässt sich ein XML Dokument verhältnismäßig einfach parsen, da der Standard von XML definierte Start und Endpunkte vorgibt in Form von öffnenden und schließenden Tags.

XML ist kein proprietäres Format, sondern wird durch das W3C entwickelt. Es kostet nichts und ist nicht an ein bestimmtes Unternehmen gebunden.

Neben JSON und CSV ist XML quasi der de facto Standard bei der Strukturierung von Daten.

In Abgrenzung zu den beiden zuvor genannten Formaten bietet XML allerdings die Möglichkeit der Selbstbeschreibung durch die Attributtags.

XML hat auch einige Nachteile, die nicht unerwähnt bleiben sollen:

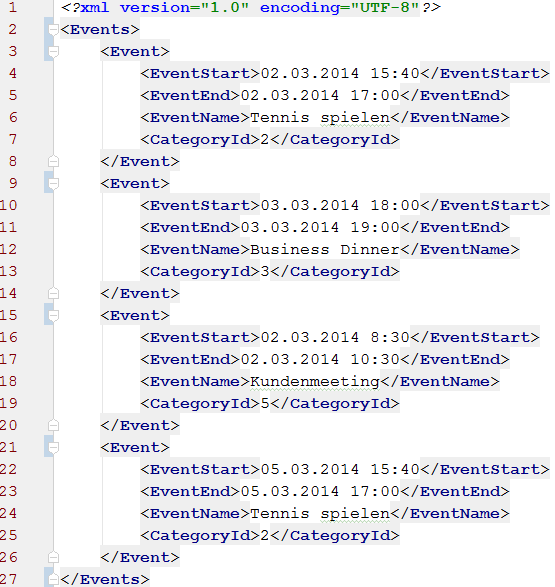
* fehlende Skalierbarkeit
* Großer Daten-Overhead
* Beschränkte Modellierungsfähigkeiten

Der Nachteil der fehlenden Skalierbarkeit und des großen Daten-Overheads durch die Tags ist aufgrund der Aufgabenstellung und der geringen Daten, die übertragen bzw importiert werden sollen, nicht relevant in Bezug auf die Aufgabenstellung. Da wir außerdem mit Strings und Integern arbeiten ist die Notwendigkeit komplexe Datentypen wie zum Beispiel Matrizen oder Tabellen abzubilden nicht gegeben.

Zusammengefasst überwiegen die Vorteile deutlich die Nachteile. Weiterhin gibt es durch die starke Verbreitung von XML viele Bibliotheken, die sich hervorragend um das Parsen von XML Dateien kümmern können.

Die XML Struktur soll dabei der Objekt Struktur der Events nachempfunden werden um die Daten direkt in das Model parsen zu können.

Eine Beispiel XML könnte so aussehen:



Es ist angedacht die XML durch Schleifen-Konstrukte pro Knotenpunkt nach Daten suchen zu lassen und bei gefundenen Daten, diese in das Eventobjekt zu schreiben. Bei falschen Datentypen oder falscher Syntax und fehlenden Daten soll eine Fehlerbehandlung eingreifen und den Benutzer informieren, dass der entsprechende Datensatz nicht importiert werden konnte.

Es macht beispielsweise keinen Sinn einer Aktivität einen Startzeitpunkt aber keinen Endzeitpunkt zuzuweisen, da ansonsten keine sinnvolle Kollisionsabfrage implementiert werden könnte.

Alle Datensätze sind Pflichtfelder, die korrekt mit Daten gefüllt sein müssen, damit das System funktionieren kann.

Bevor die Daten aus dem Eventobjekt in der Datenbank persistiert werden, soll eine Überprüfung stattfinden, ob es möglichweise Kollisionen der Zeiträume gibt. Auch in diesem Fall soll der entsprechende Datensatz nicht persistiert werden und der Benutzer über die Kollision benachrichtigt werden.

Ein Datensatz soll mit dem Knotenpunkt <Event> starten und </Event> enden und die Attribute

* EventStart
* EventName
* EventEnd
* CategoryId

beinhalten. Es soll weiterhin möglich sein mehrere Events in einer XML zusammenzufassen und importieren zu können um dem Benutzer dadurch mehr Usability zu bieten. Alternativ wäre es auch denkbar für jede Aktivität eine eigene XML bereit zu stellen, welche wir aber aus Gründen der Benutzerfreundlichkeit nicht betrachten wollen.

**Was soll die Importschnittstelle explizit nicht bieten?**

* Die importierten XMLs sollen nicht gespeichert werden
* Es sollen nicht gleichzeitig mehrere XMLs importiert werden können
* Die importierten XMLs sollen keine individuelle Dateinamen haben können

**Gedanken zur Validierung und Fehlerbehandlung**

In der Softwareentwicklung ist einer der Grundsätze niemals Benutzereingaben zu vertrauen und stets zu validieren.

Neben böswilligen Nutzereingaben, die z.B. von Hackern missbraucht werden um eigenen Code auszuführen und das System zu kompromittieren ist der weitaus größere Anteil unbeabsichtigte Eingabe von falschen Daten von unerfahrenen Benutzern, die zu ernsthaften Fehlverhalten des Programmes führen können. Da Java eine stark typisierte Sprache ist, führen falsche Datentypen zu schweren Programmabstürzen. Anders als beispielsweise bei PHP, wo die Datentypen erst zur Laufzeit automatisch generiert und gecastet werden, müssen bei Java die Datentypen schon vor dem kompilieren feststehen. Jede Übergabe von Daten muss daher abgesichert werden. Java bietet hierfür die Funktionalität von try – catch – Blöcken an um auf entsprechende Fehler zu reagieren, ohne, dass das Programm durch falsche Daten abstürzt.

Besondere Absicherung benötigen in unserer Applikation durch diese Betrachtungsweise die manuelle Eingabe der Benutzern bei den Formularen der Eventerstellung und –bearbeitung sowie bei der Categoryerstellung und –bearbeitung. Außerdem ist die Importschnittstelle eine mögliche Schwachstelle, welche besonders beachtet werden sollte.

Neben einer Validierung für Fehler durch falsche Datentypen sollte auch eine Plausibilitätsprüfung stattfinden. Da Aktivitäten eine gewisse Granularität des Tagesablaufes darstellen sollen würde es beispielweise keinen Sinn machen eine Aktivität länger als einen Tag laufen zu lassen.

**Kritische Betrachtung zum Projektabschluss**

Im Gegensatz zu der Projektplanung war es uns leider aus Zeitgründen nicht möglich all die Exceptionhandlings und Validierungen in das Projekt einfließen zu lassen, die wir uns vorgenommen hatten. In einem realen Kundenprojekt müsste an diesen Stellen auf jeden Fall nochmal nachgebessert werden und es wäre unverantwortlich ein solches System live gehen zu lassen.

Weiterhin funktioniert die Importfunktionalität noch nicht richtig, da es uns nicht möglich war eine gute Debugging- und Loggingumgebung zu realisieren um Fehler zu analysieren und auszumerzen. Insbesondere http-Requests und Sessionobjekte loggen zu können wären eine wichtige Informationsquelle zur Buganalyse gewesen.

Weiterhin wäre es wünschenswert gewesen automatisierte Tests zu haben um eine gute Softwarequalität gewährleisten zu können. Auch hier erweist sich leider die begrenzte Zeit als limitierender Faktor, da Test Driven Development eine hohe Lernkurve aufweist bevor es sich bezahlt macht. Obwohl wir schon mit Programmiervorkentnissen in PHP an das Projekt herangetreten sind, erwiesen sich doch einige Java-Eigenheiten als Fallstricke, die unerwartet viel Zeit gekostet haben.

Insbesondere zählen dazu Kompatibilitätsprobleme z.B. zwischen unterschiedlichen

JDK und Tomcat Versionen

Hibernate und Spring Versionen

JDK und Maven Versionen

Aus diesen Gründen wurden nur manuelle Tests durchgeführt und es kann nicht ausgeschlossen werden, dass nicht alle möglichen Testszenarien erfasst und überprüft wurden.

Aufgrund der vielen unterschiedlichen Versionen empfanden wir es teilweise auch als schwierig geeignete Tutorials zu finden, da sich diese doch sehr stark voneinander unterschieden haben und es als Einsteiger schwierig war einzuschätzen, welche Informationsquellen sich als hilfreich erweisen würde und welche als Zeitverschwendung.

**Maven**

Maven ist ein Dependency Management und Build Tool, welches den Workflow durch so viel Automatisierung wie möglich erleichtern und beschleunigen soll.

Es wird davon ausgegangen, dass ein Softwareprojekt immer wieder Zyklen durchläuft, wenn ein neuer Build erstellt wird. Dabei müssen nicht zwangsläufig alle Phasen durchlaufen werden, sondern es kann individuell eingestellt werden, welche Phasen benötigt werden.

Für uns wichtig waren die Phasen des validieren, kompilieren, testen, packaging und deployen.

Durch ein Mavenplugin für IntelliJ kann direkt ein DIE-interner Tomcat-Server gestartet und die kompilierten Dateien ausgeführt werden. Durch das Dependency Management von Maven ist es auch möglich die benötigten externen Bibliotheken zu laden und somit ist es einfach, das Projekt auf unterschiedlichen Systemen zu entwickeln ohne sich ständig um die Abhängigkeiten sorgen zu müssen.