it informatik

Thomas Grechenig
Mario Bernhart
Roland Breiteneder
Karin Kappel

Softwaretechnik

Mit Fallbeispielen aus realen Entwicklungsprojekten

PEARSON
Studium

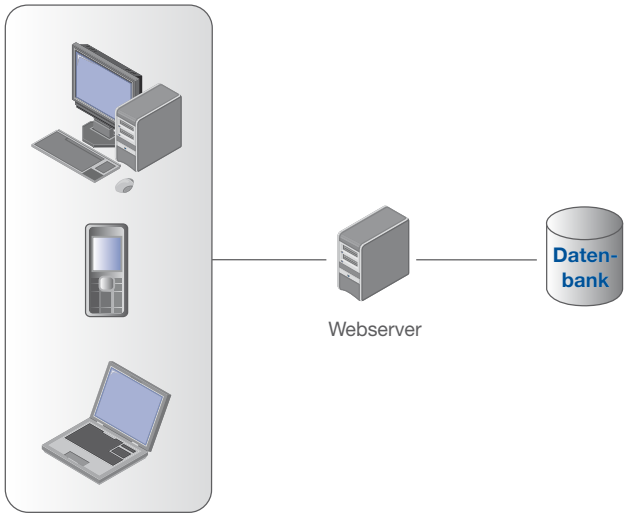

Abbildung 5.3: Der Zugriff auf Webapplikationen erfolgt mittels Webbrowser. 1..n gleichzeitige Benutzer sind erlaubt.

Als weitere Entwurfsform hat sich vor allem bei Großprojekten die *Service Oriented Architecture* (SOA) durch ihre inhärente Modularisierbarkeit etabliert. Bei der SOA wird ein Gesamtsystem in voneinander klar getrennte Komponenten aufgebrochen, die über definierte Schnittstellen mithilfe verschiedener Kommunikationsprotokolle miteinander kommunizieren können. Dabei können Komponenten auf einem einzelnen Computer installiert oder aber verteilt an verschiedenen Orten laufen (siehe ▶Abbildung 5.4). Im Normalfall erfordert die Änderung eines Geschäftsprozesses auch eine Anpassung der Softwareunterstützung – mit einem SO-System ist es jedoch möglich, die Kommunikation der Komponenten untereinander so zu ändern, dass die neue Kommunikation die erforderlichen Änderungen schnell abbilden kann. Zu diesem Zweck wurde die *Business Process Execution Language* (BPEL) entwickelt, die eine solche dynamische Konfiguration ermöglicht.

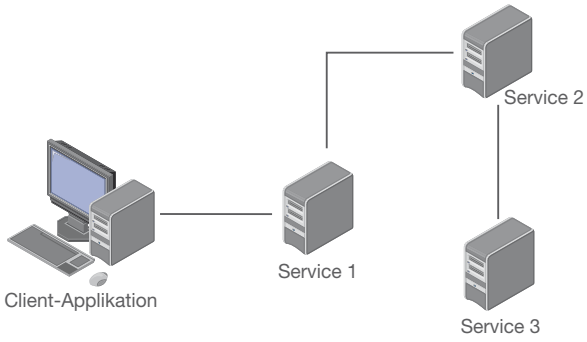

Abbildung 5.4: Schematische Darstellung einer kleinen SOA. Da die Services als komplett eigenständige Komponenten vorhanden sind, ist eine Neukonfiguration der Kommunikation mit minimalem Aufwand möglich.

Kapitel 5.1.4 beschreibt das wichtigste derzeit eingesetzte Konzept von Softwarearchitekturen.

Exkurs

Model Driven Architecture (MDA)

Das Model Driven Architecture oder Model Driven Development bezeichnet eine Entwicklungsmethode, bei der Software aus strukturierten Modellen erzeugt wird. Es wurde von der Object Management Group (OMG) im Jahr 2001 vorgestellt und stieß seitdem vor allem im akademischen Forschungsbereich auf reges Interesse, wobei es aber auch Industrie-Tools gibt. Von der abstrakten Definition bis hin zur fertigen Applikation gibt es zwei Hauptschritte:

Platform Independent Model (PIM): Dieses Modell ist unabhängig von der Zieltechnologie. Der OMG zufolge soll jede Applikation so weit abstrahierbar sein, dass es keine Abhängigkeiten von der Zielplattform gibt.

Platform specific Model (PSM): Das plattformspezifische Modell wird aus dem plattformunabhängigen Modell erstellt und in diesem Schritt bereits auf die Zieltechnologie hin angepasst. Ein Beispiel dafür ist die Abstrakte Datenbankdefinition im PIM, die bei der Transformation in ein PSM auf das Modell einer konkreten Datenbank (z.B. MySQL) hin angepasst wird.

5.1.4 Grundlegende Architekturmuster

In den vergangenen drei Jahrzehnten des Softwareentwurfs haben sich einige Grundkonzepte besonders bewährt und finden daher sowohl in Enterprise Software als auch in Applikationen für den Heimgebrauch Verwendung. Diese Konzepte basieren fast ausschließlich auf den sogenannten N-Tier-Architekturen, wobei „Tier“ übersetzt „Schicht“ bedeutet. N-Tier-Architekturen teilen Applikationen in funktional abgegrenzte Schichten ein (z.B. User Interface, Datenbank, Geschäftslogik). Dabei steht das „N“ in N-Tier-Architekturen für die Anzahl der eingesetzten Schichten. Im praktischen Einsatz sind dabei 2-Tier- und 3-Tier-Architekturen am häufigsten zu finden.

Die 2-Tier-Architektur

Abbildung 5.2 (S. 205) stellt eine 2-Tier-Architektur schematisch dar. Bei dieser exemplarischen 2-Schichten-Architektur gibt es auf der einen Seite die Anwendung, mit der der Benutzer arbeitet und die sich somit um die eigentliche Verarbeitung der Daten kümmert, und auf der anderen Seite die Backend-Komponente, bei der es sich im Standardfall um eine Datenbank handelt. Diese Art der Architektur hat den großen Vorteil, dass es eine zentral verfügbare Datenmenge gibt, die von mehreren Benutzern verwendet wird. Dabei entsteht durch die Zentralisierung des Datenbestands nur an einer einzelnen Stelle Administrationsaufwand. Der größte Nachteil besteht allerdings in der Notwendigkeit von erweiterten Maßnahmen zur Sicherstellung der *Datenkonsistenz*, da der Datenzugriff verteilt von jedem Zugriffsrechner aus erfolgt und somit in dessen Verantwortung liegt.

Die 3-Tier-Architektur

Während die Gefahr einer Dateninkonsistenz bei wenigen Zugriffen bzw. Zugriffsrechnern noch vertretbar ist, kann es spätestens bei Large-Scale-Applikationen zu massiven Problem kommen. Eine bessere Strategie für große Anwendungen ist daher die *3-Tier-Architektur*, bei der es im Unterschied zur 2-Tier-Architektur eine weitere Schicht gibt. Mit dieser Aufspaltung in drei Schichten kann die Geschäftslogik aus der Anwendungsschicht extrahiert und ähnlich wie die Datenbank zentral verfügbar gemacht werden. Die Vorteile eines konsistenten Datenzugriffs und einer besseren Wartbarkeit überwiegen hierbei den Nachteil einer längeren und detaillierteren Planungsphase. ▶Abbildung 5.5 zeigt einen konzeptuellen Aufbau einer 3-Tier-Architektur.

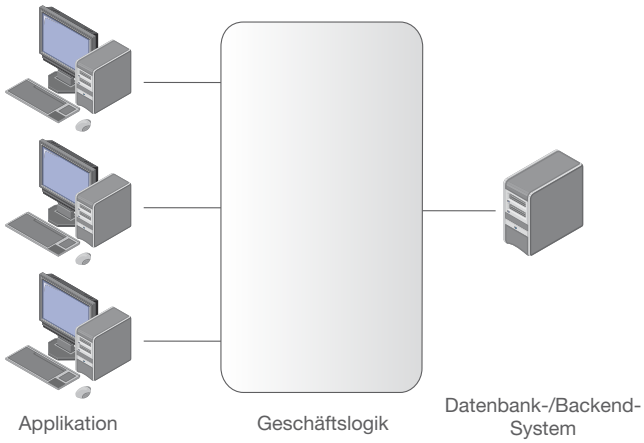

Abbildung 5.5: Eine schematische Ansicht einer 3-Tier-Architektur

Layered Architecture

Wie oben erwähnt bestehen die n-Tier-Architekturen aus funktionell getrennten Ebenen (Tiers). Aber auch diese Tiers sind auf einem konkreteren Level wiederum in Schichten geteilt, die sogenannten Layer.

Die Layer sind dabei hierarchisch aufgeteilt und folgen in einem sauber designten System bestimmten Regeln:

- Eine Ebene kommuniziert nur mit der Ebene direkt darunter.
- Eine Ebene kommuniziert nicht mit den darüber liegenden Ebenen.
- Eine Ebene hat keine Abhängigkeiten auf einer der darüber liegenden Ebenen.
- Die Kommunikation zwischen den Ebenen erfolgt über definierte Interfaces, um den Austausch einer Ebene zu ermöglichen.
- Auftretende Exceptions müssen nach Ebene gekapselt werden.

Wenn diese grundlegenden Regeln eingehalten werden, verbessert dies die Wartbarkeit und etwaige Fehlerbehebungen. Durch die Austauschbarkeit von Ebenen kann so flexibel auf Anforderungsänderungen reagiert werden.

Bei der praktischen Implementierung bedeutet das meist, dass jede Ebene durch ein bestimmtes Framework implementiert wird (siehe Kapitel 6).

▶Abbildung 5.6 zeigt den schematischen Aufbau einer Schichtenarchitektur.

Abbildung 5.6: Schematischer Aufbau einer Schichtenarchitektur. Die Kommunikation erfolgt nur nach unten, da Kommunikation nach oben das Konzept einer Schichtenarchitektur verletzen würde.

Fallstudie

Zur Architektur von Lisame

Das Projekt Lisame ist ein gutes Beispiel für die Umsetzung einer Schichtenarchitektur. Wenn man den Webapplikationsteil näher betrachtet, kann man ganz klar drei Schichten erkennen:

- User-Interface: Das User-Interface wurde mittels Java Server Faces (JSF) realisiert. JSF ist ein Framework, das eine schnelle und effiziente Implementierung einer Weboberfläche ermöglicht.
- Geschäftslogik: Die Geschäftslogik basiert auf einem Enterprise Java Beans Framework.
- Datenbankzugriff: Der Datenbankzugriff wurde über das Hibernate Frame gelöst. Hibernate bietet die Möglichkeit, mit minimalem Aufwand einen Zugriffslayer zu bauen, der über Konfigurationsdateien auf verschiedene Szenarien anpassbar ist, ohne den eigentlichen Quellcode ändern zu müssen
- Um die Kommunikation zwischen den einzelnen Frameworks zu homogenisieren, wurde zusätzlich schichtenübergreifend das Metaframework JBoss Seam gewählt.

Pipes and Filters

Pipes and Filters bezeichnet eine Architekturform, die vor allem bei der Verarbeitung von Datenströmen zum Einsatz kommt, wobei die Verarbeitung der Daten in mehreren Schritten geschieht und es zum Entwicklungszeitpunkt noch nicht klar ist, ob diese Schritte immer in derselben Reihenfolge durchlaufen werden sollten bzw. ob in Zukunft (mittelbar oder unmittelbar) neue Zwischenschritte in die Verarbeitung integriert werden sollen.

Das zu erschaffende System muss also einen Strom aus Input-Daten verarbeiten. Ein entsprechendes System als eine einzige Komponente zu implementieren, kann aus mehreren Gründen nachteilig sein:

- Das System (die einzelnen Verarbeitungsschritte) wird von mehreren Entwicklern entwickelt.
- Die Verarbeitung der Daten wird in mehrere Bearbeitungsstufen aufgegliedert.
- Die Anforderungen zu den Stufen werden sich wahrscheinlich ändern.

Um diesen Problemen vorzubeugen, sollten also die Bearbeitungsschritte austauschbar und in ihrer Reihenfolge veränderbar sein. Durch diese Flexibilität kann eine ganze Familie an Systemen erstellt werden, die existierende, verarbeitende Komponenten verwendet. Daraus ergeben sich gewisse Anforderungen an den Entwurf des Systems.

Durch eine *Pipes and Filters*-Architektur wird die Aufgabe eines Systems in sequentielle Bearbeitungsschritte aufgeteilt. Die Schritte sind durch den Datenfluss durch das System miteinander verbunden. Jeder Bearbeitungsschritt wird in Form einer Filterkomponente implementiert. Ein Filter konsumiert und liefert Daten inkrementell, um eine niedrige Verzögerung und parallele Verarbeitung zu ermöglichen. Die Datenquelle, die Filter und die Datensammelstelle *(Data Sink)* sind durch Pipes sequentiell miteinander verbunden. Eine Pipe implementiert einen Datenfluss zwischen zwei benachbarten Verarbeitungsschritten. Die Sequenz von Filtern und Pipes wird als *processing pipeline* bezeichnet (Buschmann et al., 1996).

Fallstudie

Zur Architektur von Stadt21

Wie bereits in der Fallbeispielbeschreibung in Kapitel 3 erwähnt, sollte im Rahmen von Stadt21 ein Tool erstellt werden, das die Angestellten der Stadtverwaltung bei der Erledigung der täglichen Aufgaben optimal unterstützen sollte. Während der Anforderungsanalyse zeigte sich, dass jede dieser Tätigkeiten aus einer genau definierten Abfolge von Arbeitsschritten besteht. Somit sollten im neuen System Workflows dynamisch definiert werden, die dann für die Benutzer ohne Neukompilierung und erneutes Ausrollen des Systems verfügbar sind. Daher wurde Stadt21 in zwei Hauptkomponenten aufgeteilt, die über einen zentralen Server abgerufen werden können:

- *Arbeitsoberfläche:* Die Benutzerschnittstelle für das Tagesgeschäft wurde als Webapplikation realisiert (JSP mit ICEFACES). Mit den gewählten Komponenten war es möglich, die Eingabefunktionalität einer Rich-Client-Anwendung in einer Webapplikation abzubilden. Durch die Implementierung als Webapplikation sind die Benutzer außerdem nicht an ihren Arbeitsplatz gebunden.
- *Prozesseditor:* Der Prozesseditor ist das Werkzeug, um neue Prozesse zu definieren. Er wurde als Java-Anwendung implementiert, da Funktionalität benötigt wurde, die sich zum Entwicklungszeitpunkt nur ungenügend über Webapplikations-Frameworks bereitstellen ließ. Um die Applikation trotzdem im gesamten Netzwerk verfügbar zu machen, sollte deren Aufruf per *Java Web Start* erfolgen. Bei Java Web Start handelt es sich um einen Mechanismus, um Java-Applikationen über den Webbrowser zu starten (siehe ▶Abbildung 5.7).

- *Serverkomponente:* Wie schon erwähnt wurde die Präsentationsschicht mit JSP realisiert, während für die Geschäftslogik EJBs zum Einsatz kamen. Die Kommunikation mit der Datenbank wurde mittels eines Persistancy Layer auf Hibernate Basis umgesetzt.

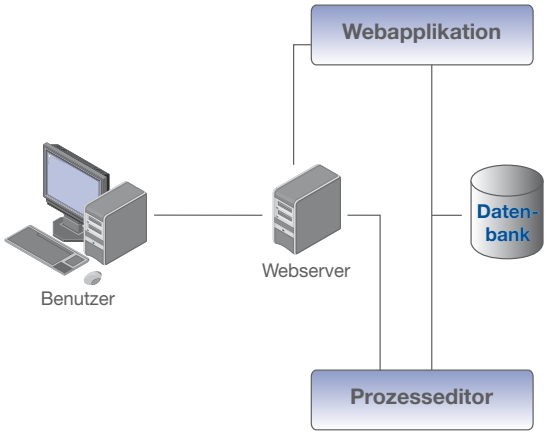

Abbildung 5.7: Schematische Darstellung der Stadt21-Architektur

Fallstudie

Zur Architektur von Touch&Go

Da bei Touch&Go zahlreiche Technologien zum Einsatz kommen, ist die architektonische Perspektive besonders interessant (siehe ▶Abbildung 5.8). Bei Touch&Go geht es um eine Ticketing-Lösung im Mobilfunkbereich auf NFC-Basis. Da bei diesem Projekt das bestehende System nicht ersetzt werden sollte, wurde im Entwurf viel Wert auf eine nahtlose Integration in die bestehende Umgebung gesetzt.

Im Wesentlichen gibt es bei Touch&Go zwei Rollen, den *Endkunden*, der über sein Mobiltelefon auf das System zugreift, und den *Administrator,* für den eine eigene administrative Sicht auf das System existiert. Der Zugriff auf das System ist für den Benutzer des Mobiltelefons nur durch Informationen auf der Mobiltelefon-Applikation ersichtlich, die über eine drahtlose Netzwerkverbindung mit dem Touch&Go-Server (*Ticket Server*) kommuniziert. Daher ist eine grafische Benutzerschnittstelle für diese Benutzergruppe nicht erforderlich. Für die Administration des Systems wurde hingegen eine grafische Schnittstelle auf Basis von Java Server Pages geschaffen.

Als *technologische Basis* wurde Java gewählt, da nach ersten Analysen eine Lösung basierend auf *Enterprise Java Beans* (EJB) am vielversprechendsten erschien. Als Serversystem wurde der JBoss Application Server gewählt, für die Mobiltelefon-Anwendung J2ME und für die eigentliche Kartenapplikation Java Card.

Der Zugriff auf die Datenbank erfolgt mittels einer Persistenzschicht (Hibernate), dies ermöglicht einen Datenbankwechsel (z.B. von PostgreSQL zu Oracle) nur durch Änderung der Hibernate-Konfiguration.

EJBs bieten die Möglichkeit, ein System in unabhängige Komponenten (Beans) zu teilen, die sowohl innerhalb einer Jboss-Instanz als auch über Instanzgrenzen hinweg miteinander kommunizieren können. Dieses Feature war von enormer Wichtigkeit sowohl für die Lastverteilung als auch für die Anbindung an das bestehende System. Konkret werden bei Touch&Go folgende Enterprise-Java-Technologien verwendet:

- *Session Beans:* Session Beans sind Komponenten, die durch einen äußeren Einfluss instanziiert und aktiviert werden. Session Beans können je nach Bedarf „stateful" oder „stateless" sein. Stateless Beans speichern Informationen nur während eines Aufrufes und verwerfen diese nach Verbindungsende wieder (d.h., jeder Clientaufruf erzeugt eine neue Instanz), Stateful Beans dagegen behalten für jeden Client Informationen, die durchaus mehrere Verbindungen überdauern können (vgl. Warenkorb in einem beliebigen Webshop).
- *Message-Driven Beans* (MDB): MDBs werden an eine Warteschlange (Message-Queue) gebunden. Wenn in diese Warteschlange eine Nachricht (Message) gesendet wird, wird die MDB aktiviert. Nach dem Verarbeiten einer Nachricht wird die MDB wieder in einen Schlafzustand versetzt und bleibt bis zur nächsten Nachricht inaktiv.
- *Web Services:* Für die Kommunikation zum bestehenden System wurde ein Entwurf basierend auf Web Services erstellt. Die Kommunikation zwischen der neu implementierten Web-Service-Schnittstelle und dem bestehenden System erfolgt über SOAP über HTTPS. SOAP ist ein Kommunikationsstandard, der von verschiedenen Entwicklungs-Frameworks unterstützt wird, gut dokumentiert ist und dadurch einen raschen Entwicklungsvorgang versprach.
- *Java-Server-Pages (JSP):* Für die Webapplikations-Komponente wurden JSPs eingesetzt, da dies eine schnelle Entwicklung aufgrund existierender Erfahrung ermöglichte.

Im Weiteren wurden auf Seiten des Mobiltelefons noch zwei weitere Java-Technologien eingesetzt, um den Kommunikationsverlauf zu vervollständigen:

- *Java Midlet:* Auf Basis von J2ME wurde eine Mobiltelefonanwendung entwickelt, welche die Kommunikation zwischen dem Secure Element und dem Ticket Server ermöglicht.
- *Java Card Applet:* Mittels Java Card wurde eine Smart-Card-Applikation entwickelt. Auf dieser Smart-Card-Applikation werden sämtliche Informationen zu den gekauften und entwerteten Tickets verwaltet.

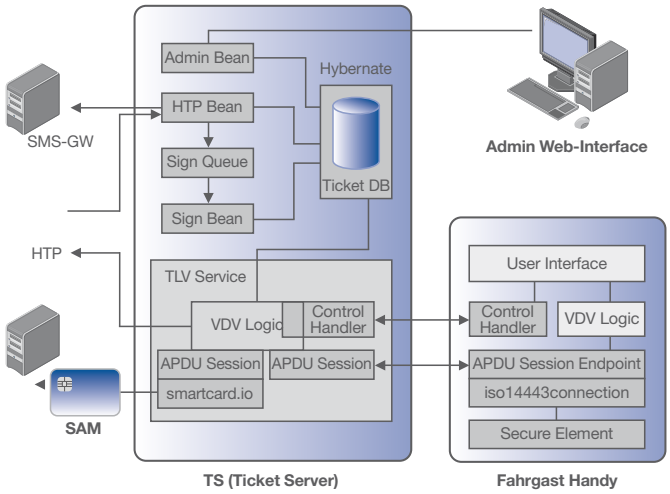

Abbildung 5.8: Architektur von Touch&Go

Fallstudie

Zur Architektur von HISS

Das Fallbeispiel HISS beschreibt eine Softwarelösung für ein universitäres Backend. Als Aufgabenstellung wird ein veraltetes, historisch gewachsenes System ersetzt und die notwendigen Schnittstellen werden homogenisiert. Das alte System ist dabei über Jahrzehnte gewachsen und wurde in Anpassung an neue gesetzliche Vorgaben laufend verändert. Das neue System HISS sollte als einziges zentrales System universitätsweit eingesetzt werden (siehe ▶Abbildung 5.9). Daraus entstanden die folgenden Kernanforderungen:

- *Rollenbasiertes Benutzermanagement:* Das System muss verschiedene Sichten für die verschiedenen Nutzergruppen bieten (Studenten, Lehre, Forschung, Administration).
- *Webapplikation:* Um einen ortsunabhängigen Zugriff zu ermöglichen, wurde HISS als Webapplikation geplant.
- *Hohe Verfügbarkeit:* Da der gesamte Universitätsbetrieb auf das neue System umsteigen sollte, musste eine hohe Verfügbarkeit gewährleistet werden.
- *Performance:* Da es erfahrungsgemäß gerade zu Semesterbeginn (Lehrveranstaltungsanmeldungen) zu erhöhter Systemlast kommt, musste das System performant genug sein, um eine solche Last zu bewältigen.
- *Anbindung an Legacy-Systeme:* Da es Subsysteme gab, die durch das neue System nicht ersetzbar waren, mussten Schnittstellen geschaffen werden, um eine Kommunikation zu diesen Legacy-Systemen zu ermöglichen.

Als technologische Basis für HISS wurde Ruby on Rails (RoR) gewählt. RoR ist ein schlankes Webapplikations-Framework basierend auf dem Model-View-Controller (MVC)-Paradigma. RoR erlaubt im Gegensatz zu PHP-basierten Weblösungen eine strenge Trennung zwischen Geschäftslogik und Darstellungsschicht. Als Datenbank wurde die bestehende Oracle-Datenbank verwendet, um eine aufwändige Datenmigration zu vermeiden.

Durch die Neuimplementierung des Systems konnten schon während der Entwicklung Performanz-Steigerungen von ca. 250% beobachtet werden.

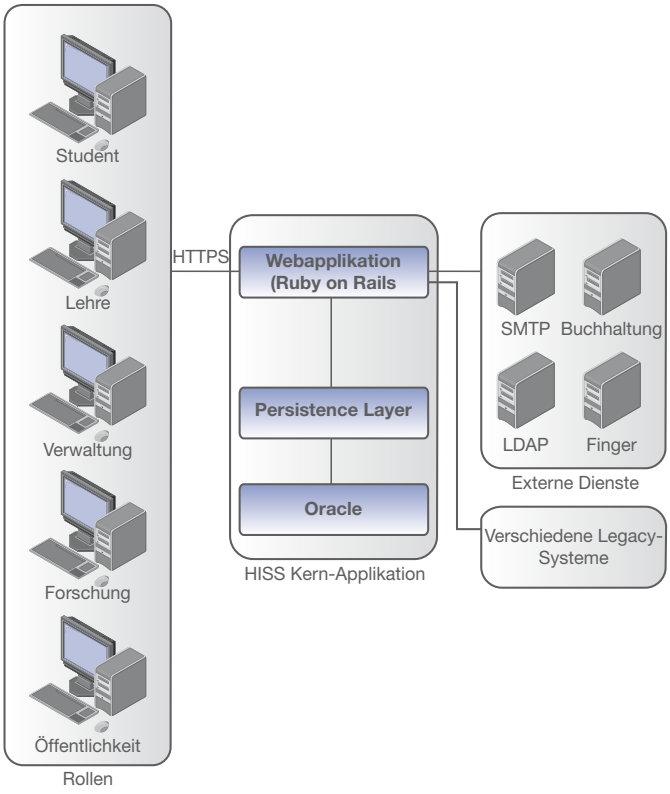

Abbildung 5.9: Schematische Darstellung der HISS-Architektur. Die Kommunikation des Benutzers mit dem System erfolgt über HTTPS. Mit Legacy-Systemen (zum Teil noch in Cobol) wird mittels XML-Datei-Austausch kommuniziert.

5.1.5 Architekturbausteine und deren Kommunikation

Jede Architektur besteht auf der Makroebene aus Bausteinen (*building blocks*). Diese Bausteine sind für unterschiedliche Aufgaben verantwortlich und kommunizieren untereinander. An dieser Stelle werden einige gängige Architekturbausteine beschrieben und anschließend typische Kommunikationsmuster erläutert. In den meisten Ausprägungen

von interaktiven Systemen werden Architekturbausteine grundsätzlich nach dem *Client/Server*-Muster unterschieden. Dabei wird davon ausgegangen, dass ein Server oder Service von mehreren Clients verwendet wird und die Kommunikation von Seiten des Clients ausgeht. Auf relevante architekturbeeinflussende Infrastrukturdienste, z.B. Loadbalancer (*Lastverteiler)* oder *Firewalls*, wird an dieser Stelle nicht weiter eingegangen. Zu den gängigsten generischen serverbasierten Architekturbausteinen zählen:

- Webserver
- Portalserver
- Applikationsserver
- Datenbankserver
- Nachrichtenserver (Message Queue), Kommunikationshub
- Dokumenten-Management-System (DMS)
- Transaktionsserver
- Verzeichnisserver (Directory)
- Stapelverarbeitung (Batch Processing)
- Public Key Infrastructure (PKI), siehe Kapitel 13

Neben den generischen Bausteinen gibt es in jeder Domäne spezifische Bausteine (z.B. ein *Clearing & Billing Server*), die in der jeweiligen Domäne von Relevanz sind.

Aufgabe – Architekturbausteine Wählen Sie ein Szenario von der CWS und entscheiden Sie, welche Architekturbausteine Sie für eine Implementierung benötigen. Diskutieren Sie Alternativen und begründen Sie Ihre Entscheidung.

Daten

Die Kommunikation zwischen einem Client und einem Server oder zwischen den Servern spielt für die Architektur eine entscheidende Rolle. Die Kommunikation kann u.a. unter folgenden Gesichtspunkten charakterisiert werden:

- *Direktionalität*: Die Kommunikation zwischen zwei Komponenten kann entweder unidirektional oder bidirektional erfolgen. Auch in der bidirektionalen Kommunikation ist oft nur ein Kommunikationspartner der Auslöser (z.B. Request/Response).
- *Zustandsbasiertheit*: Die Kommunikation kann entweder zustandsbasiert (*stateful*) oder zustandslos (*stateless*) erfolgen. In der zustandsbasierten Kommunikation erhält der Kommunikationspartner Information über die Kommunikation während der gesamten Kommunikationsdauer, während in der zustandslosen Kommunikation diese Information üblicherweise anstatt im Service in den Nachrichten gekapselt ist.
- *Zugriffsart*: Die wichtigste Zugriffsart ist das *Request/Response*-Muster. Des Weiteren wird häufig zwischen einer *Push*-Kommunikation (Senden von Information) und einer *Pull*-Kommunikation (Abholen von Information) unterschieden.
- *Synchronität*: Die Kommunikation kann im Request/Response-Muster entweder *synchron* (der Sender wartet, oftmals blockierend, auf die Antwort nach Senden der Anfrage) oder *asynchron* (der Sender setzt die Programmausführung nach Senden der Anfrage fort und verarbeitet die Antwort ereignisgesteuert) erfolgen.

- *Standardkonformität*: Die Kommunikation kann konform zu standardisierten offenen oder proprietären Protokollen erfolgen. Es gilt als fundamentale best practice, grundsätzlich nach offenen Standards zu kommunizieren.
- *Absicherung*: In vielen modernen Infrastrukturen wird die Kommunikation zwischen Architekturbausteinen (z.B. kryptografisch) gesichert. Die Sicherung kann entweder Teil des Protokolls selbst sein oder das Protokoll wird zur Sicherung in ein entsprechendes Trägerprotokoll gekapselt (z.B. https).

Eine Grundregel für einen stabilen Schnittstellenentwurf ist „*be strict in what you send and be liberal in what you accept*“. Der Verwender der Schnittstelle muss also, so weit es ihm möglich ist, die ausgehenden Informationen prüfen und nur eine strikte konforme Kommunikation nach außen zulassen. Der Schnittstellenanbieter wiederum muss im Sinne eines stabilen Gesamtsystems alle möglichen (und vielleicht sogar nicht spezifizierten) Eventualitäten der Schnittstellenverwendung berücksichtigen und geeignet verarbeiten. Ein typisches Beispiel für diesen Fall ist das sogenannte *trailing spaces*-Problem. In vielen Kommunikationsprotokollen sind die Feldlängen nicht begrenzt. Ein Verwender einer Schnittstelle könnte nach einem Datum noch Leerzeichen mitsenden, ohne die Konformität zur Schnittstellenbeschreibung zu verletzen. Der Empfänger könnte wiederum beim *Parsing* eine feste Zeichenlänge erwarten und in diesem Fall einen Ausnahmefehler erzeugen. Gemäß dem oben genannten Prinzip müssten beide Kommunikationspartner ein Entfernen der unnötigen Leerzeichen (*trim*) vornehmen. Wenn die Kommunikation nicht *peer-to-peer,* sondern über einen geeigneten *Kommunikationshub* (Message Orientated Middleware, MOM) implementiert wird, werden solche Prüfungen und Anpassungen üblicherweise vom Hub durchgeführt. Hieraus leitet sich auch ein weiterer bekannter Satz ab: „*Every distributed application developer (DAD) needs a message orientated middleware (MOM).*“

Exkurs

Asynchronous Javascript and XML (AJAX)

Asynchronous Javascript and XML (*AJAX*) ist ein asynchrones Kommunikationsdesign für Webanwendungen. Um nicht bei jedem Ereignis (üblicherweise eine Benutzerinteraktion) den vollen und teilweise sehr umfangreichen http-*Request/Response-Zyklus* durchzuführen, wird nur das Ereignis an eine entsprechende Serverkomponente gesendet. Die asynchrone Antwort enthält dann nicht wie im klassischen Modell üblich die gesamte Webpage, sondern nur die Information, die für ein Update erforderlich ist. ▶Abbildung 5.10 zeigt die Funktionsweise von AJAX. Im Webportal des Fallbeispiels LiSaMe werden viele AJAX-Komponenten (diese kapseln dieses Kommunikationsmuster) verwendet, um die Benutzbarkeit und das Antwortzeitverhalten der Applikation zu optimieren.