2.1. Java EE

Die Grundlage der technischen Umsetzung von MeDIC, sowie zahlreicher weiterer in der Forschungsgruppe Softwarekonstruktion entwickelter Projekte [13, 49, 45], bildet eine Sammlung von Technologien, deren Definition und Interoperabilität in der Spezifikation von *Java Platform, Enterprise Edition*[1] (Java EE) zusammengefasst sind [52]. Im Mittelpunkt steht die Entwicklung verteilter, mehrschichtiger Webanwendungen mit dem Fokus auf Geschäftsanwendungen. Dazu verfolgt Java EE einen komponentenorientierten Ansatz und spezifiziert in modularer Weise eine Reihe von Server- und Client-Architekturen. Diese bilden ein Rahmenwerk und unterstützen den Entwickler bei typischen wiederkehrenden Aufgaben und Anforderungen (siehe Abschnitt 2.1.1 und 2.1.2). Ein ausgeglichenes Programmiermodell mit Annotations- und Konfigurations-Management ermöglicht zudem große Flexibilität bezüglich Anpassungen bestehender Komponenten oder Erweiterungen durch eigene Komponenten. Dabei folgt Java EE dem Grundsatz *Konvention vor Konfiguration*, um den Quelltext so kompakt und übersichtlich wie möglich zu halten.

Als Laufzeitumgebung für die erstellte Webanwendung dient ein sogenannter Anwendungserver, welcher die von der Spezifikation definierten Funktionalitäten implementiert. Dazu gehören unter anderem Sicherheitsanforderungen, Session-Management, Resource Injection, Persistierung oder Transaktionsmanagement. Es existieren zahlreiche zertifizierte Implementierungen für einen Java-EE-Anwendungserver [35]. Diese umfassen sowohl Open Source als auch kommerzielle Projekte. Die Forschungsgruppe Softwarekonstruktion setzt auf den *GlassFish*-Anwendungsserver – der Open-Source-Referenzimplementierung, welche von der Oracle Corporation unterstützt wird.

Der klassische Grundaufbau einer Java-EE-Anwendung ist in Abbildung 2.1 zu sehen. Auf oberster Ebene befindet sich die Client-Schicht auf der Seite des Anwenders. Üblicherweise[2] läuft hier ein Webbrowser, welcher die dynamischen Webinhalte (XHTML, Grafiken, ...) darstellt und ausführt (JavaScript), die vom Server empfangen wurden. Die Client-Schicht wird manchmal auch mit dem Begriff *Thin Client* betitelt. Damit will man verdeutlichen, dass diese Schicht möglichst wenig Applikationslogik implementiert. Komplexe Geschäftsoperationen werden serverseitig durchgeführt – für gewöhnlich in der sogenannten Business-Schicht von den *Enterprise JavaBeans*. Dazu zählen auch Datenbankabfragen oder die Kommunikation mit anderen Systemen.

2.1.1. Enterprise JavaBeans

Enterprise JavaBeans (kurz EJB) kapseln die Geschäftslogik einer Java-EE-Applikation und bilden somit das Herzstück der Client-Server-Architektur. Kontextuell befinden

[1] aktuell in Version 6

[2] Die andere Möglichkeit ist ein dedizierter Application Client.

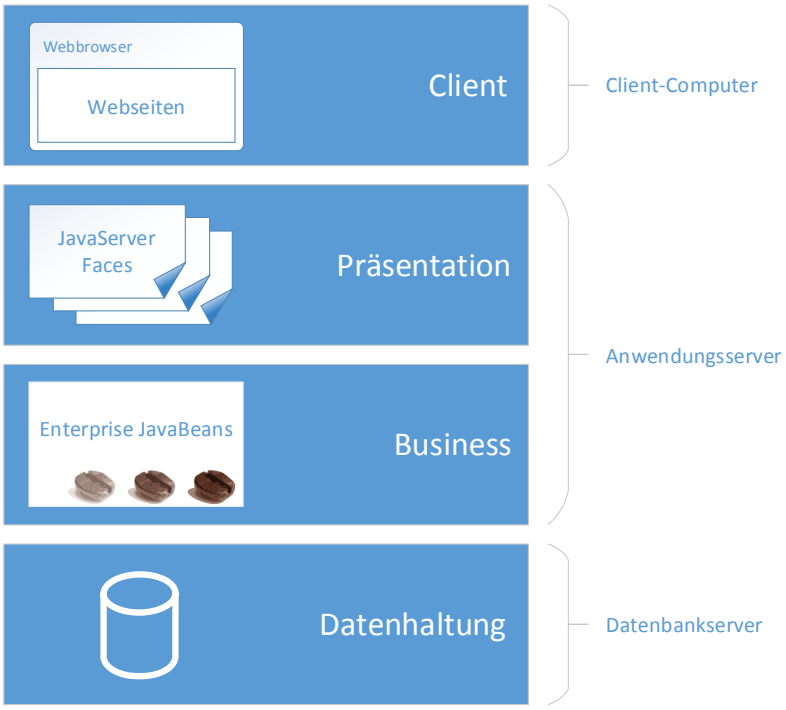

Abbildung 2.1.: Grundaufbau einer Java-EE-Anwendung (nach [38])

sie sich in der Business-Schicht, welche auf der Datenhaltungsschicht aufsetzt. Die Präsentationsschicht befindet sich eine Ebene höher und ist verantwortlich für die Generierung der Ausgabedaten, die schließlich zum Client gesendet werden (siehe Abschnitt 2.1.2). EJB-Komponenten setzen sich aus einfachen Java-Klassen und -Interfaces zusammen – den sogenannten POJOs (Plain Old Java Objects) und POJIs (Plain Old Java Interfaces)[3]. Durch diese einfach gehaltene Mikroarchitektur ist es möglich beliebige Klassenmodelle auf EJB-Komponenten zu übertragen. Wie bereits eingangs erwähnt, wird die gesamte Webanwendung (oder Webapplikation) innerhalb einer speziellen Laufzeitumgebung ausgeführt, die der Java EE Server bereitstellt. Bezogen auf die Enterprise JavaBeans ist das ein sogenannter EJB-Container, welcher u.a. folgende Dienste bereitstellt:

- Java Persistence API (JPA) zur Persistierung von Java-Objekten
- Java Transaction API (JTA) für integriertes Transaktionsmanagement
- Java Message Service (JMS) zur Kommunikation zwischen Java EE Komponenten nach dem Publish-Subscribe Entwurfsmuster oder nach dem Punkt-zu-Punkt Modell
- Rollenbasierte Zugriffsbeschränkung für Klassen und Methoden als

[3] aktueller Stand nach EJB Version 3.1

Sicherheitskonzept

- Contexts and Dependency Integration (CDI) zur externen Konfiguration von Abhängigkeiten
- und viele mehr

Dadurch wird dem Entwickler ein signifikanter Teil infrastrukturellen Codes abgenommen mit dem Ziel, die Entwicklungszeiten zu verkürzen und die Qualität der Applikationen zu steigern.

Die EJB-Komponenten werden nach dem *Inversion of Control* Prinzip per *Dependency Injection* [19] eingebunden. Der Funktionsumfang reicht vom Lese-Zugriff auf einfache Daten-Objekte bis hin zu vollständiger CRUD-Logik (Create, Retrieve, Update, Delete) mit erweiterten Geschäftsfunktionen.

2.1.2. JavaServer Faces

JavaServer Faces (kurz JSF) ist eine Rahmenwerk-Technologie zur Generierung von XHTML-basierten Benutzeroberflächen. Der GlassFish-Anwendungsserver liefert mit *Mojarra* die offizielle Referenzimplementierung mit. Der folgende Abschnitt behandelt die Leistungsmerkmale von JavaServer Faces 2.1. Den Hauptbestandteil von JSF bildet eine XHTML-basierte Auszeichnungssprache [63], die *View Declaration Language*, mit der sich sogenannte *Facelets* definieren lassen [36]. Jene lassen sich aus speziellen JSF-Komponenten zusammensetzen, welche

- entweder aus einer Menge von integrierten Standardkomponenten stammen,
- Teil einer externen Komponentenbibliothek sind (z.B. PrimeFaces [42] oder JBoss RichFaces [43]),
- deklarativ als benutzerdefinierte Composite-Komponente definiert wurden
- oder eine Java-Implementierung der abstrakten Klasse *UIComponent* mit eigenem Renderer sind.

Während die Seitendeklaration der Facelets die Vorlage für die XHTML-Ausgabe darstellt, schaffen speziell annotierte Java-Klassen, die *Managed Beans*, eine Verbindung zu den Attributen und Methoden der EJB-Komponenten. Managed Beans sind einfache Java-Klassen (POJOs), die per `@ManagedBean` gekennzeichnet sind.

Abbildung 2.2 skizziert die Beziehung der verschiedenen Java-EE-Komponenten zueinander. Ein Facelet kann eine oder mehrere Managed-Beans referenzieren. Konkret werden Eigenschaften oder Methoden im Managed-Bean referenziert, die den JSF-Komponenten als Datenmodell dienen oder zur Ereignisbehandlung benötigt werden. Die Managed-Beans erhalten wiederum Zugriff auf die im Klassenrumpf deklarierten EJBs, welche per Dependency Injection vom Web-Container des

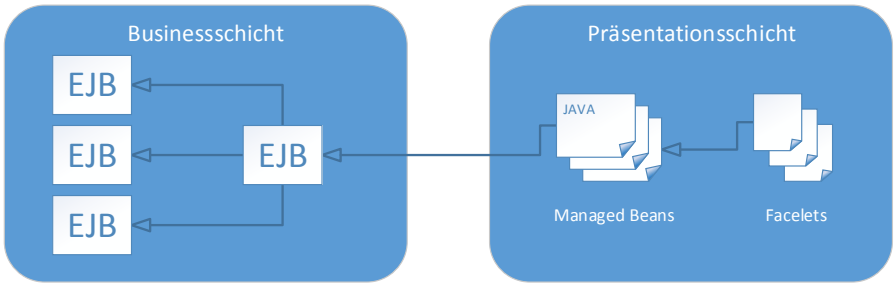

Abbildung 2.2.: Komponenten-Beziehungen von EJB und JSF

Anwendungsservers bereitgestellt werden. Auf diese Weise lässt sich eine klare Trennung von Geschäftslogik und Präsentation realisieren (siehe dazu Abschnitt 2.1.4). Die EJB-Komponenten lassen sich untereinander gleichermaßen verknüpfen und wiederverwenden. Ein oft verwendetes Entwurfsmuster ist die Verwendung einer Fassade, die Funktionalität über eine einheitliche und vereinfachte Schnittstelle zu einer Menge von EJB-Komponenten bereitstellt.

Der Java-EE-Webcontainer instanziiert die Managed-Beans zur Laufzeit und generiert in Verbindung mit den dazugehörigen Facelets die XHTML-Ausgabe. Die Lebensdauer eines Managed-Bean wird ebenfalls per Annotation festgelegt[4] und ermöglicht damit die Zustandsspeicherung über die Laufzeit einer Sitzung hinweg. Genauer gesagt bleibt die Instanz einer Managed-Bean über mehrere aufeinanderfolgende HTTP-Anfragen erhalten. Selbige wird für den Garbage Collector freigegeben, sobald die Sitzung abgelaufen ist (Timeout) oder durch den Benutzer beendet wurde (z.B. Logout oder Session-Cookie gelöscht). Die Spezifikation sieht sechs vordefinierte *Scopes* vor, welche die Lebensdauer eines Managed-Bean festlegen:

- `@RequestScoped`:
  Das Managed-Bean bleibt für die Zeitdauer einer HTTP-Anfrage bestehen. Es wird erzeugt, sobald die Anfrage den Server erreicht hat und bei der Generierung des Komponentenbaums benötigt wird (siehe Abschnitt 2.1.3). Nachdem die Anfrage vollständig abgearbeitet und die Antwort generiert wurde, wird das Managed Bean zerstört.
- `@ViewScoped`:
  Die Instanz des Managed-Bean bleibt bestehen, solange der Benutzer mit der gleichen Ansicht (im Browserfenster) interagiert. Sobald der Benutzer eine neue Ansicht ansteuert, wird das Managed Bean zerstört.
- `@SessionScoped`:
  Das Managed-Bean bleibt für die Lebensdauer einer Sitzung bestehen. Das heißt so lange der Benutzer mit der Anwendung interagiert - möglicherweise über

[4] Die Konfiguration der Managed-Beans ist aus historischen Gründen ebenfalls per *faces-config.xml* möglich.

mehrere Ansichten hinweg. Nach einer bestimmten Zeit[5] ohne Interaktion durch den Benutzer, wird die Sitzung automatisch beendet.

- `@ApplicationScoped`:
  Das Managed-Bean existiert über die gesamte Lebensdauer der Anwendung hinweg genau einmal – unabhängig von Benutzeraktionen.
- `@NoneScoped`:
  Das Managed-Bean wird jedes mal neu erzeugt, wenn es intern bei der Generierung des Komponentenbaums referenziert wird. Der Nutzen dieses Scopes erschließt sich aus speziellen Anwendungsszenarien auf die hier nicht näher eingegangen werden soll [48].
- `@CustomScoped`:
  Die Lebensdauer des Managed-Beans kann der Entwickler selbst bestimmen, indem dessen Instanziierung und Zerstörung eigenhändig durchgeführt wird.

2.1.3. Lebenszyklus von HTTP-Anfragen

Die JSF-Spezifikation definiert einen sogenannten *Lebenszyklus* (engl. lifecycle), der für jede HTTP-Anfrage durchlaufen werden muss. Am Ende des Lebenszyklus steht die Antwort, die zurück an den Client geschickt wird. Bei der Antwort kann es sich entweder um ein vollständiges XHTML-Dokument handeln oder um eine partielle Antwort (engl. partial response), um einen bestimmten Teil der Webanwendung zu aktualisieren. Wie in Abbildung 2.1.3 zu sehen, gibt es insgesamt sechs Phasen, die nacheinander abgearbeitet werden. Es werden jedoch nicht immer alle Phasen durchlaufen – in bestimmten Fällen sind Abkürzungen möglich. Zunächst muss zwischen zwei Anfragearten unterschieden werden:

1. **Initiale Anfrage**:
   Wird eine Ansicht zum ersten Mal von einem Benutzer angesteuert, muss aus der Seitendeklaration des Facelets als erstes ein sogenannter Komponentenbaum erzeugt werden. Dies geschieht in der *Restore-View*-Phase.

   Alle JSF-Komponenten, die in dem jeweiligen Facelet zum Einsatz kommen, werden in einem solchen Baum organisiert und dienen als Grundlage für alle Operationen der darauffolgenden Phasen.

   Bei einer initialen Anfrage folgt auf Phase 1 direkt die *Render-Response*-Phase, in der aus dem Komponentenbaum und dem damit verknüpften Datenmodell die XHTML-Antwortseite erzeugt wird.
2. ***Postback*-Anfrage**:
   Eine sogenannte Postback-Anfrage tritt genau dann auf, wenn Formulardaten

[5] z.B. 30 Minuten; die Standardeinstellung ist Anwendungsserver-spezifisch

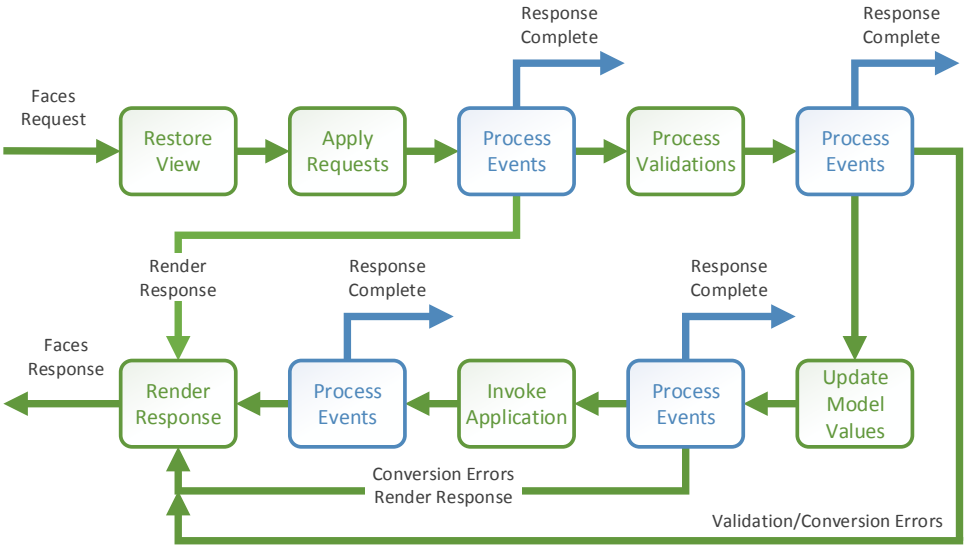

Abbildung 2.3.: Der Lebenszyklus von HTTP-Anfragen in JSF (nach [37])

per HTTP POST [17] Befehl an den Server gesendet werden. In diesem Fall müssen nach der Generierung des Komponentenbaums in Phase 1 folgende Phasen durchlaufen werden.

- *Apply Request Values*: In dieser Phase werden die übertragenen Parameter aus der HTTP-Anfrage extrahiert und im Komponentenbaum an entsprechender Stelle als vorläufig zwischengespeichert.
- *Process Validations*: Die zwischengespeicherten Werte werden in dieser Phase in entsprechende Java-Objekte konvertiert und gleichzeitig validiert.

  Schlägt die Konvertierung oder Validierung fehl, werden dazugehörige Fehlermeldungen generiert und im Komponentenbaum angehängt. Anschließend folgt auf direktem Wege die Render-Response-Phase, in der die Antwortseite mit den zuvor generierten Fehlermeldungen erstellt wird.
- *Update Model Values*: Nach erfolgreicher Konvertierung und Validierung wird in dieser Phase schließlich das hinterlegte Datenmodell aktualisiert, indem die Setter (dt. Änderungsmethoden) der entsprechenden Objekte aufgerufen werden.
- *Invoke Application*: In dieser Phase können bestimmte, an Ereignisse der jeweiligen Komponente gebundenen Funktionen des Managed-Beans aufgerufen werden. Zusätzlich ist es möglich über Steuerkomponenten[6] die

[6] bspw. `h:commandButton` oder `h:commandLink`

Navigation mit Hilfe des Rückgabewertes einer sogenannten Action-Funktion zu beeinflussen.

- *Render Response*: Schließlich wird in der letzten Phase des Lebenszyklus die Antwortseite generiert, indem der Komponentenbaum durchlaufen wird und die Render-Funktion jeder einzelnen Komponente aufgerufen wird. Das Ergebnis ist entweder ein vollständiges XHTML-Dokument oder im Fall einer Ajax-Anfrage eine *Partial-Response*.

  Ajax[7] steht für asynchrone Datenübertragung zwischen Browser und Server und ermöglicht dem Client das Nachladen von Daten im Hintergrund. JSF macht sich dieses Konzept zu Nutze um Teile der Internetseite zu aktualisieren. Ajax-Anfragen sind in JSF immer Postback-Anfragen und liefern ein spezielles XML-Dokument, welches nur die Ausgabe der zu aktualisierenden Komponenten enthält.

Dieser kurze Überblick über den Lebenszyklus soll dem Leser einen Einblick in die Konzepte von JSF geben und zum besseren Verständnis, der in der Realisierung (Kapitel 7) benutzten Technologien beitragen. Die Arbeitsweise von JSF ist weitaus komplexer und vielfältiger als hier beschrieben. Abbildung 2.3 deutet weitere Abbruchfälle der einzelnen Phasen an.

2.1.4. Model View Controller

Das *Model-View-Controller*-Prinzip (MVC) steht für ein Konzept zur Strukturierung von Software-Artefakten in die drei Einheiten *Model* (dt. Datenmodell), *View* (dt. Repräsentation oder Ansicht) und *Controller* (dt. Programmsteuerung). Ursprünglich 1979 konzipiert für den Entwurf von Benutzeroberflächen in der Programmiersprache Smalltalk [44], ist es heute elementarer Bestandteil moderner Webentwicklung. Jede bedeutende Programmiersprache für serverseitige Webentwicklung [60] unterstützt heutzutage das Konzept, welches eine klare Trennung der Zuständigkeiten ermöglicht und dadurch die Wiederverwendung und Erweiterbarkeit der einzelnen Komponenten erhöht.

Folgende Auflistung gibt einen kurzen Überblick über die sechs populärsten [60] serverseitigen Programmiersprachen zur Webentwicklung und die dazugehörigen MVC-Rahmenwerke:

1. **PHP**: Technisch möglich durch Rahmenwerke von Drittanbietern wie etwa *Symfony* [14] oder *TYPO3 Flow* [57]
2. **ASP.NET**: Nativer Bestandteil in Form von *ASP.NET MVC* [34].
3. **Java**: Realisiert in Form des Komponenten-Rahmenwerks JavaServer Faces.

[7] Asynchronous JavaScript and XML

4. **ColdFusion**: Baut ebenfalls auf Erweiterungen von Drittanbietern (z.B. Coldbox [39]).
5. **Perl**: Drittanbieter (z.B. Catalyst [4]).
6. **Ruby**: Fester Bestandteil von Ruby on Rails [26].

Mit der Aufnahme von JavaServer Faces 1.2 in den Java-EE Standard [52] im Jahre 2006 hat Sun Microsystems[8] eine stärkere Modularisierung bei der Entwicklung der Benutzerschnittstelle vorangetrieben. Erstmals bestand eine echte Trennung von Inhalten, deren Repräsentation und der Programmsteuerung.

- *Model*: Das Datenmodell enthält die darzustellenden Daten. Bezogen auf JSF sind das die Attribute eines Managed-Bean, welche in Form von EJB-Komponenten auch die zugrundeliegende Geschäftfslogik enthalten.
- *View*: Die Ansicht ist eine grafische Repräsentation des Datenmodells. Bei JSF entspricht diese dem generierten XHTML-Dokument, welches im Webbrowser beim Client dargestellt wird. Die strukturelle Vorlage der Repräsentation bilden die Facelets.
- *Controller*: Die Programmsteuerung ist für die Ereignisbehandlung zuständig. Sie beobachtet Benutzereingaben und kann daraufhin Aktionen auslösen, die das Datenmodell und die Ansicht verändern. Typische Ereignisse einer JSF-Komponente sind z.B. `valueChange` oder `action` (Schaltfläche betätigen). Genau genommen sind Teile der Programmsteuerung in der Client-Skriptsprache JavaScript implementiert und werden clientseitig ausgeführt. Andere Teile befinden sich im Managed-Bean und werden in der *Invoke Application*-Phase aufgerufen.

Generell ist eine klare Aufteilung der Software-Artefakte in die drei Einheiten *Model*, *View* und *Controller* nicht immer offensichtlich. Insbesondere bei der Geschäftslogik ist sowohl eine Zuordnung zum Datenmodell möglich als auch die Zuordnung zur Programmsteuerung.

2.2. Webservice

Ein Webservice bildet eine Schnittstelle zum Datenaustausch lose gekoppelter heterogener Systeme. Die Kommunikation erfolgt dabei über ein Netzwerk bzw. das Internet. Wie im klassischen Paradigma der *serviceorientierten Architektur* (SOA) [32] verteilter Systeme gibt es auch beim Webservice die Rollen des Konsumenten und des Dienstleisters.[9] Letzterer realisiert Dienstzugangspunkte (end points) in Form von Netzwerkadressen, welche durch URIs[10] spezifiziert werden, über die potentielle

[8] heute Oracle Corporation
[9] Auch Diensterbringer und Dienstnehmer genannt [24].
[10] Uniform Resource Identifier [1]