jsf-academy.com

# UI-Architekturen mit JavaServer Faces

**Andy Bosch**
**JSF-Academy.com**
**Stuttgart**

**Schlüsselworte**

Architektur, User Interface, JavaServer Faces (JSF)

## Einleitung

User Interface Architekturen - das ist für viele bereits ein Widerspruch in sich. Warum spricht man über Architekturen im UI? Sind da nicht nur bunte Pixel zu finden? Die "echten" Architekturentscheidungen treffen doch wohl "richtige" Entwickler, also die, die sich mit Datenbanken und der Service-Integration befassen. Aber UI-Entwickler ...?

Dass diese Annahmen Nonsens sind, zeigt diese Session. Es müssen sehr wohl wichtige Architekturentscheidungen gerade im UI-Layer getroffen werden. Die Entscheidung zum Aufbau des User Interface betreffen zudem auch andere Schichten.

Architekturen sind grundsätzlich unabhängig von einer gewählten Architektur. Allerdings helfen Technologien durchaus, eine gute Architektur zu realisieren. Speziell beim Einsatz von JavaServer Faces können bestimmte Pattern optimal umgesetzt werden. Der Artikel wird somit auf generelle Pattern in UI-Architekturen eingehen, sich speziell aber auf den Einsatz von JSF beziehen.

## Warum sind UI-Architekturen überhaupt wichtig?

Wären die ersten beiden Buchstaben (UI) nicht in der Überschrift enthalten, würde jeder natürlich der Aussage sofort zustimmen, dass eine gute Software-Architektur ganz entscheiden für den Erfolg eines Projektes ist. Jetzt gibt es natürlich keine "reine" User-Interface Architektur, da auch ein UI Layer als Teil des Gesamtsystems, und damit als Teilbereich der Gesamtarchitektur, betrachtet werden muss.
Es gibt aber durchaus einige Muster und Vorgehensweisen, die sich in (Groß-) Projekten speziell beim Aufbau des UI-Layers bewährt haben. Ziel ist es schließlich, das UI-Layer derart aufzubauen, dass es sich harmonisch in die Geamtarchitektur integriert.

## Über Design Prinzipien zu Design Patterns bis zu einer Architektur

Doch wie beginnt man, eine UI-Architektur zu konzipieren? Das Problem der weißen Seite kennen kennt jeder Buch-Autor. Der Anfang ist immer das Schwerste. Ähnlich geht es auch einem Software-Architekten, wenn es heisst: "Jetzt erstelle doch mal eine tragfähige (UI-) Architektur".
Hier kann man sich zunächst auf die grundlegenden Erkenntnisse der Software-Entwicklung berufen. Design Prinzipien wurden schon vor vielen Jahren etabliert und sind natürlich heute noch genauso aktuell wie "damals". Wobei man auch sagen muss, dass sich regelmässig neue Prinzipien etablieren. Für unseren Anwendungsfall, der Architektur im User-Interface Bereich, könnte man Prinzipien wie

Separation of Concerns als Beispiel voranstellen. Ok, ganz allgemeine Prinzipien wie DRY (Don't repeat yourself) passen natürlich auch immer.

Die nächste Evolutionsstufe wären Design Patterns, die sehr viel konkreter bestimmte Problemstellungen erörtern und Lösungsvorschläge anbieten. Design Patterns sind von der Grundidee her programmiersprachen-neutral, doch gibt es die Klassiker der Design Patterns auch schon konkret z.B. in Java umgesetzt. Passend für das UI wären u.a. Patterns wie Model-View-Controller (MVC) oder auch Model-View-Presenter (MVP).

Wir haben nun einige Bausteine und können sie in eine gewisse Struktur bringen. Um das Thema der UI Lösungsansätze einzugrenzen, nehmen wir als Arbeitshypothese, dass wir insgesamt eine Drei-Schichten-Architektur umsetzen wollen (vgl. Abb. 1).

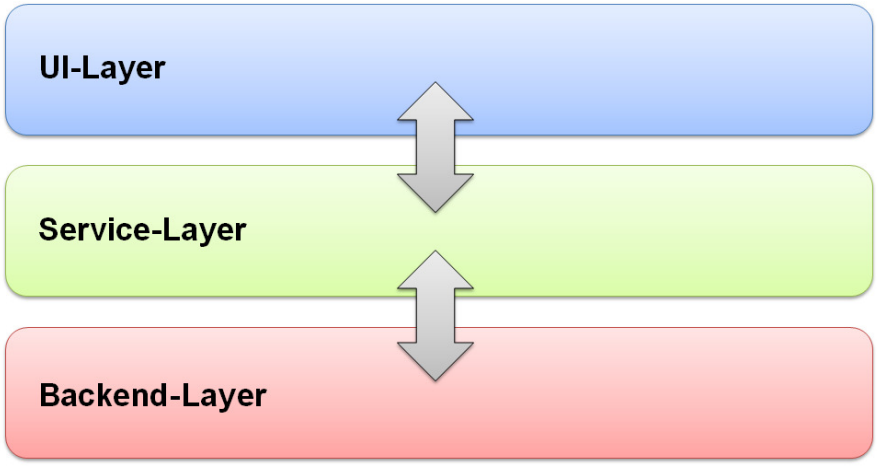


*Abb. 1: Klassische Drei-Schichten-Architektur*

**Model-View-Controller und JSF - geht das?**

Ok die theoretischen Grundlagen haben wir. Wir möchten im User Interface Prinzipien wie Separation of Concerns berücksichtigen und z.B. Model-View-Controller umsetzen. Doch die JSF-Spezifikation erwähnt keines dieser Stichworte. In JSF haben wir lediglich Managed Beans. Nicht selten finde ich in Projekten genau diese Situation vor. Managed Beans sind für die Datenhaltung zuständig, beinhalten aber gleichzeitig Funktionalität. Ein klarer Verstoß gegen SoC.
Daher die Empfehlung, hier eine strikte Trennung von Model und Controller einzuführen. Ich spreche in diesem Zusammenhang gerne von Model Managed Beans und Controller Managed Beans. Mit dieser sehr einfachen logischen Trennung läßt sich ein hervorragendes MVC umsetzen.

**Der Übergang von UI zum Service**

Wichtig für die Einordnung des MVC-Patterns in die Gesamtarchitektur ist das Verständnis, dass MVC hier Teil des UI-Layers ist. Das M ist zunächst ein Viewmodell, das C ein UI-Controller (und eben kein fachlicher Controller). An bestimmten Punkten in der Anwendung findet somit ein Übergang aus dem UI-Layer in den Service-Layer statt. Hierzu kann man sich in JSF verschiedene technische Möglichkeiten vorstellen: Aus einem ActionListener, in einer Aktionsmethode, bei der

Validierung, während des Modellupdates oder sogar in Getter-Methoden. Sehr schnell kann dies ausarten, und man hat keinen kontrollierten Übergang von UI in den Service mehr (vgl. Abb. 2).

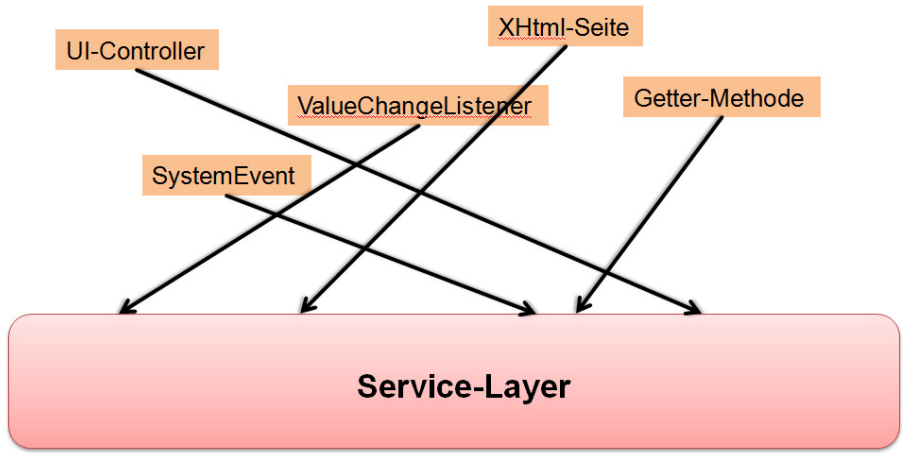


*Abb. 2: Service-Aufrufe aus dem UI*

Dieser Ansatz ist (zumindest für größere Projekte) nicht zu empfehlen. Es sollte ein kontrollierter Aufruf stattfinden, meine Empfehlung lautet hier: Service-Aufrufe nur aus dem UI-Controller.
Der Ansatz, lediglich aus dem UI-Controller einen Service-Übergang zuzulassen, hat viele Vorteile. So können an den entscheidenden Punkten Performance-Messungen vorgenommen werden oder auch ein Mocking der Service-Schicht aufgebaut werden.

**Last but not least: Das M in MVC**

Eine Frage, die schon fast religiös diskutiert wird: Ist das M im UI-Layer ein reines UI-Bean, oder nicht eher ein allgemeines Domänenobjekt, das im UI lediglich zur Anzeige kommt. Gerade mit aktuellen Persistenz-Technologien wie JPA oder Hibernate liegt es nahe, den POJO-basierten Ansatz aus der Persistenz-Schicht bis in die UI-Schicht durchzuziehen. Kann nicht eine Person, die per JPA geladen wird, als Model-Managed-Bean in der UI-Schicht verwendet werden?

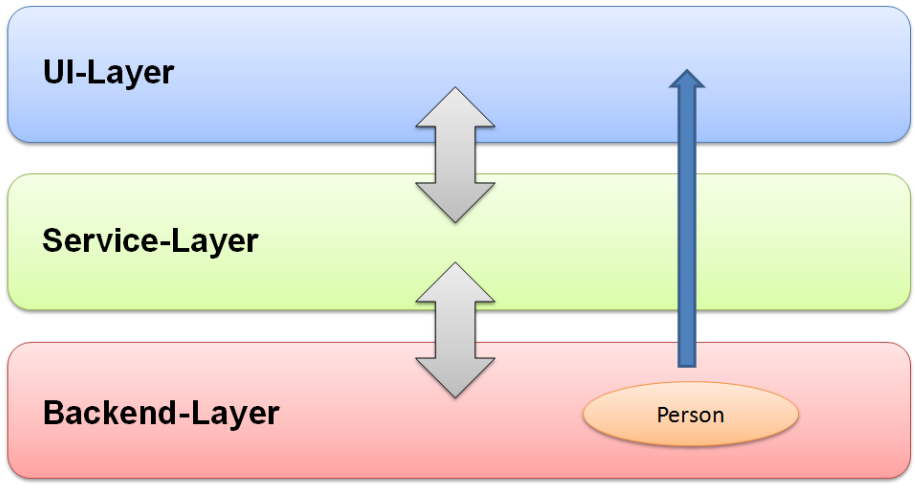


*Abb. 3: Wo liegt das Modell?*

Wenn man als Arbeitshypothese annimmt, dass eine Datenbank-Entität bis in das UI hochgereicht wird, ist die nächste Frage, ob wir auf attached oder detached Entities arbeiten. Jeder dieser Ansätze hat Vor- und Nachteile. Würde man auf detached Entities arbeiten, hätte man das Problem, dass Objekte ggf. schon als komplett geladen in das UI hochgereicht werden müssten. Stellen sie sich eine Person vor, die eine Relation auf Adressen hat. Natürlich möchte man im UI darüber navigieren, also Aufrufe wie myPerson.getAdressen() sollten schon funktionieren. Würde man Entitäten aber immer komplett laden, könnte das eventuell mit der großen Nachteilen verbunden sein. Je nach Komplexität der Datenbankstruktur kann bei vollständigem Laden in manchen Projekten die gesamte Datenbank eingelesen werden. Kein so ein optimaler Ansatz.

"Dann arbeite doch auf attached Entites" lautet folglich die Empfehlung. Man reicht ein Objekt an das UI, und wenn das UI auf Properties zugreift, die noch nicht geladen sind, werden diese einfach nachgelesen. Hat dann noch irgendjemand Kontrolle über die erzeugten SQLs?

Der klassische Ansatz wäre, dann eben auf DTOs (Data Transfer Objects) zu setzen. Zwischen dem UI und der Persistenz gibt es nochmals eine Stelle, die ein Mapping vornimmt und dem UI reine Beans für das Frontend bereitstellt. Natürlich hat man hier den Nachteil, dass ein Mapping stattfinden muss,

Wie fast immer im Leben, muss man gewisse Kompromisse eingehen. Jeder dieser Ansätze hat verschiedene Vorteile, aber auch Nachteile. Daher muss für jedes Projekt erneut im Architektenkreis diskutiert werden, welcher Ansatz der Beste ist.

**Kontaktadresse:**

**Andy Bosch**
JSF-Academy.com
Barchetstraße 19
D-70569 Stuttgart

| | |
|---|---|
| Telefon: | +49 (0) 711 - 46 97 28 70 |
| Fax: | +49 (0) 711 - 88 25 321 |
| E-Mail | andy.bosch@jsf-academy.com |
| Internet: | www.jsf-academy.com |