***MVC Architektur & Templates***

*Kompakthandbuch*

ckg-logo.png

*Matthias Jenzen, CKG Entwicklung*

Inhalt

1 Überblick 3

1.1 Ein Rückblick 3

1.2 Überarbeitung MVC Design Architektur (Ziel 1) 3

1.3 Pflege MVC Komponenten + HTML Templates (Ziel 2) 3

2 MVC Design Architektur 4

2.1 Erweiterung um Hierarchie + Vererbung 4

2.1.1 Deklaration der Theme Hierarchie 4

2.1.2 Bestehende Templates ersetzen (ab einer Theme Ebene) 4

3 MVC Komponenten + HTML Templates 6

3.1 Kapselung bestimmter Web Komponenten 6

3.1.1 Umsetzung 6

3.2 Vereinfachung Grid Komponente 6

3.2.1 Umsetzung „Vorher“ / „Nachher“ 7

# Überblick

## Ein Rückblick

**Nach mehreren Jahren erfolgreichen Einsatzes der neuen Web Technologie „MVC“ benötigen wir eine Überarbeitung MVC Architektur, sowie einiger MVC Komponenten + HTML Templates.**

**Hintergrund ist, dass unsere MVC Architektur stets anspruchsvollsten Kundenanforderungen z. B. nach flexiblen Web Designs in unterschiedlichster Optik + verändertem HTML Seitenaufbau gerecht werden muss.  
Hierbei ist die größte Herausforderung, alle neuen Web Portale mit ihren unterschiedlichen Designs stets auf nur einer einzigen Code Basis bereitstellen zu können.   
Denn nur mit einer einzigen Code Basis sind geringste Wartungsaufwände sowie effiziente Funktionserweiterungen mit maximaler Produktivität überhaupt möglich.**

**Zudem ist die neue Architektur historisch schnell gewachsen und muss(te) dabei trotzdem den ständig neuen Anforderungen des Web gerecht werden.   
Im historischen Verlauf waren aber nicht immer Zeit + Budgets vorhanden, alle neuen Komponenten so sauber in die Architektur zu integrieren, dass ein maximal schlanker Programmier Code – verbunden mit geringstem Einarbeitungsaufwand insbesondere für neue / externe Entwickler – erzielt werden konnte.**

***Unsere MVC Architektur muss stets anspruchsvollsten Kundenanforderungen z. B. nach flexiblen Web Designs in unterschiedlichster Optik + verändertem HTML Seitenaufbau gerecht werden!***

## ****Überarbeitung MVC Design Architektur (Ziel 1)****

Seit 2015 setzen wir in unserer Architektur erstmals mehrere Designs ein.  
Um die Web Templates künftig maximal redundanzfrei halten zu können, muss unsere Web Design Architektur (Design Themes) auf die Funktionen „Hierarchie + Vererbung“ erweitert werden.

Durch die Bereitstellung dieser hierarchischen Vererbungsarchitektur können wir künftig die Design Basis für neue Web-Portale frei wählen und nur die nötigen Abweichungen im Design implementieren.

## Pflege MVC Komponenten + HTML Templates (Ziel 2)

Für einige HTML Komponenten und -Templates wird bei jeder Wiederverwendung unnötig redundanter .NET Code bzw. Javascript- / HTML-Code „mitgeschleppt“.  
Hier besteht nun der dringende Handlungsbedarf, den betroffenen Code in neue Standardbausteine zu verlagern oder bei bestehenden Bausteinen in die tieferen Schichten dieser Komponenten zu kapseln, um die stark erschwerenden Code-Redundanzen zu vermeiden, die letztendlich Wartbarkeit und Lesbarkeit des gesamten Codes immens erschweren.

# ****MVC Design Architektur****

Die Zielanforderungen im Detail:

## Erweiterung um Hierarchie + Vererbung

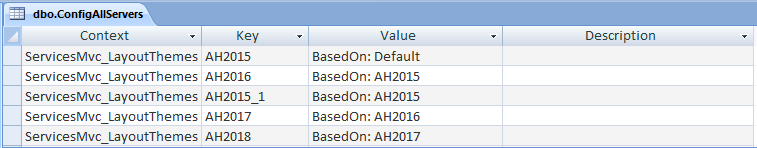
Die neuen MVC Web Design Themes müssen um die Funktionen „Hierarchie + Vererbung“ erweitert werden.   
Beispiel, konkreter Bedarfsfall aus der Praxis:

Neben dem Standard Metronic Design Theme des DAD setzen wir seit Kurzem das neue „Kroschke On“ Design Theme ein.   
Das „On“ Theme soll nun die Basis für das Erscheinungsbild künftiger Web Portale stellen, wie z. B. das „Kroschke COSTAR“ Portal.

Hierzu benötigen wir zwingend die Möglichkeit einer hierarchischen Ableitung aus bestehenden Themes, also quasi ein Feature, bestehende Funktionen + Design-Features vom Basis Theme (Kroschke On) zu erben und nur die Abweichungen im Ziel Portal (Co Star) zu „überschreiben“ bzw. neu zu implementieren.

Durch die Bereitstellung dieser hierarchischen Vererbungsarchitektur können wir künftig die Design Basis für neue Web-Portale frei wählen und nur die nötigen Abweichungen bezüglich Optik und Seitenaufbau mit geringstem Aufwand sowie ohne Code Redundanz implementieren.

### Deklaration der Theme Hierarchie

Zunächst müssen die Themes definiert und ihre Erbhierarchie deklariert werden.  
Dies geschieht über eine simple Datenbanktabelle.  
Da alle Themes auf allen Portalen + Servern zum Einsatz kommen können (aber nicht müssen), verwenden wir hier unsere Tabelle „ConfigAllServers“ die in unserer Entwicklungsdatenbank (ähnlich SAP „CKD“) gepflegt wird und über einen automatischen Sync-Prozess auf alle System transportiert wird:  
  
Neben unseren beiden Produktiv Themes „default“ und „AH2015“ beschreiben wir noch ein paar weitere Beispiel-Themes.  
Die Daten der Tabelle lassen erkennen, dass das Autohaus ON Theme „AH2015“ direkt vom „default“ Theme (= Metronic Theme für DAD MVC Anwendungen) abgeleitet ist.  
Zwei weitere Themes leiten sich dann von „AH2015“ ab („AH2015\_1“ und „AH2016“).  
Vom Theme „AH2016“ vererbt sich dann successiv ein weiteres Theme auf einer jeweils tieferen Hierarchieebene bis „AH2018“.

### Bestehende Templates ersetzen (ab einer Theme Ebene)

Bereits umgesetzt und produktiv ist die Möglichkeit des Ersetzens bestehender Templates ab exakt der obersten Theme Ebene (default Theme) bis zur exakt ersten Unterebene (AH2015 Theme).  
Neu ist nun das Feature, Templates „ab“ einer beliebigen Hierarchie zu ersetzen.  
  
Beispiel:  
Eine Template wurde in Theme AH2015 vom Default Theme ersetzt (geht schon).  
Nun kann diese Template z. B. im Theme AH2017 neu definiert werden und ersetzt somit die korrespondierenden Templates aus den Themes „default“ und „AH2015“.  
Die neue Template ist „ab“ 2017 gültig, also laut unserer Hierarchiedefinition für die Themes „AH2017“ + „AH2018“ (und ggfls. weitere Erbfolgen, wenn denn so definiert).

# MVC Komponenten + HTML Templates

Die Zielanforderungen im Detail:

## Kapselung bestimmter Web Komponenten

Jedes Mal wenn die betroffenen Web Komponenten (z. B. Multi-Select, etc) zum Einsatz kommen, entsteht unnötig redundanter HTML + CSS Code (partiell auch Script Logik), der die Wartung bestehender Einsatzfälle sowie den erneuten Einsatz dieser Komponenten unnötig verkompliziert.

Ziel ist es, für die betroffenen Komponenten wiederverwendbare Programmierbausteine zu entwickeln, die idealerweise künftig mit nur einer einzigen Codezeile zum Einsatz kämen.

### Umsetzung

## Vereinfachung Grid Komponente

Unsere hauseigene Grid Komponente ist nicht nur der definitiv zentrale + mächtigste Baustein innerhalb unserer Web Komponentensammlung, er kommt auch am häufigsten zum Einsatz, z. B. in unseren Standard Web-Reports und in unseren Assistenten gesteuerten Anwendungen („wizard driven applications“).

Der ursprüngliche Quellcode unserer Grid-Komponente ist Open Source und wurde ab dem Stand 2013 von uns in eigener Regie weiter entwickelt.

***… die Anforderungen unserer Kunden an unsere Grid Komponente sind immens …***

Die Anforderungen seitens unserer Kunden an unsere Grid Komponente sind dabei immens:  
Fähigkeiten wie kundenbezogene Spaltennamen + Lokalisierungen, ein Wysiwyg Übersetzungs-Feature für (Kunden-)Administratoren, sowie eine maximal verzögerungsfreie Darstellung, schnelle serverseitige Filterung und auch das Caching von großen Datenmengen - auch bei langsamen Internetverbindungen - sind nur eine Auswahl der inzwischen verfügbaren Features.

Die Features „Caching“ + „Filterung“ bringen allein im .NET Backend eine Ansammlung von ca. 2 Bildschirmseiten Code pro Grid Einsatz mit sich, für die wir uns wünschen, dass sie einmalig + zentral gekapselt werden.  
Auch auf den Front End Seiten haben wir es unnötiger Weise mit einer gewissen Anzahl wiederkehrender Script Zeilen zu tun.

Hier besteht nun der dringende Handlungsbedarf, den betroffenen Code in einem Standardbaustein zu kapseln.   
Dieser Code ist war nicht per se redundant, seine Lesbarkeit und somit seine Wartbarkeit leiden aber darunter in nicht unerheblichem Maße, dass es insbesondere neuen bzw. externen Entwicklern die Einarbeitung stark erschwert.

### Umsetzung „Vorher“ / „Nachher“

Im Folgenden wird der Umbau der Grid Komponente an einem Beispiel farblich verdeutlicht:

Ursprünglicher Code, der nicht mehr benötigt wird, aber selbstverständlich aus Gründen der Abwärtskompatibilität in anderen Sourcen bestehen bleiben kann, wird auskommentiert und abgeblendet dargestellt:   
// Beispiel für alten, nicht mehr benötigten Code

Neuer Code bzw. neue Funktionalität wird gelb hervorgehoben:   
Beispiel für neue Funktionalität  
 **View** Schicht, Code Beispiel:

Html.XTelerik()

.XGrid<StrafzettelModel>(controllerName: "Strafzettel", dataSource: "Strafzettel",

multiSelect: false/true,

multiSelectHint: "Bitte 1 oder mehr Strafzettel wählen“)

.Name("GridStrafzettel")

.Columns(columns => columns.XBoundAll())

.XSort(sortOrder => sortOrder.Add(c => c.VertragsNr).Descending())

.XAutoGridCreate()

//.XAutoAjaxDataBinding()

//.XAutoColumnConfiguration()

//.ClientEvents(events => events.XAutoCommonClientEvents())

//.XPageSize(10)  
 //.XToolBar()

**Controller** Schicht, Code Beispiel:

Erfreulicherweise fällt in der Controller nur Code weg!  
Das heißt, dass die Komplexität deutlich vermindert und die Lesbarkeit des Codes erleichtert wird!

***Hier fällt nur Code weg! 🡺 Höhere Lesbarkeit des verbleibenden Codes***

//[HttpPost]

//public JsonResult StrafzettelStrafzettelSelectionChanged(string key, bool isChecked)

//{

// if (key.IsNullOrEmpty())

// ViewModel.SelectStrafzettel(isChecked);

// else

// ViewModel.SelectStrafzettel(key, isChecked);

// return Json(new { allSelectionCount = ViewModel.SelectedStrafzettel.Count });

//}

//[GridAction]

//public ActionResult StrafzettelAjaxBinding()

//{

// //return View(new GridModel(ViewModel.StrafzettelFiltered));

// return View(new GridModel(ViewModel.FilteredItemsGetFor("Strafzettel")));

//}

//[HttpPost]

//public ActionResult FilterGridStrafzettel(string filterValue, string filterColumns)

//{

// ViewModel.FilterStrafzettel(filterValue, filterColumns);

// return new EmptyResult();

//}

//protected override IEnumerable GetGridExportData()

//{

// //return ViewModel.StrafzettelFiltered;

// return ViewModel.FilteredItemsGetFor("Strafzettel");

//}

**Model** Schicht, Code Beispiel:

Auch hier fällt eine Menge Code weg, nur 1 neue Code Zeile kommt hinzu!

***🡺 Deutlich weniger Komplexität durch Einsparung von Code!***

public void DataMarkForRefresh()

{

//PropertyCacheClear(this, m => m.StrafzettelFiltered);

ShadowCopyClearFor(this, m => m.Strafzettel);

}

//[XmlIgnore]

//public List<EquiPartlistSelektor> SelectedStrafzettel

//{

// get { return Strafzettel.FindAll(e => e.IsSelected); }

//}

//public void SelectStrafzettel(string vin, bool select)

//{

// var fzg = Strafzettel.FirstOrDefault(f => f.FahrgestellNr == vin);

// if (fzg == null)

// return;

// fzg.IsSelected = select;

//}

//public void SelectStrafzettel(bool select)

//{

// StrafzettelFiltered.ForEach(f => f.IsSelected = select);

//}

//[XmlIgnore]

//public List<StrafzettelModel> StrafzettelFiltered

//{

// get { return PropertyCacheGet(() => Strafzettel); }

// private set { PropertyCacheSet(value); }

//}

//public void FilterStrafzettel(string filterValue, string filterProperties)

//{

// StrafzettelFiltered = Strafzettel.SearchPropertiesWithOrCondition(filterValue,

// filterProperties);

//}