

.NET: ASP.NET MVC

© J. Heinzelreiter
Version 1.4

Überblick

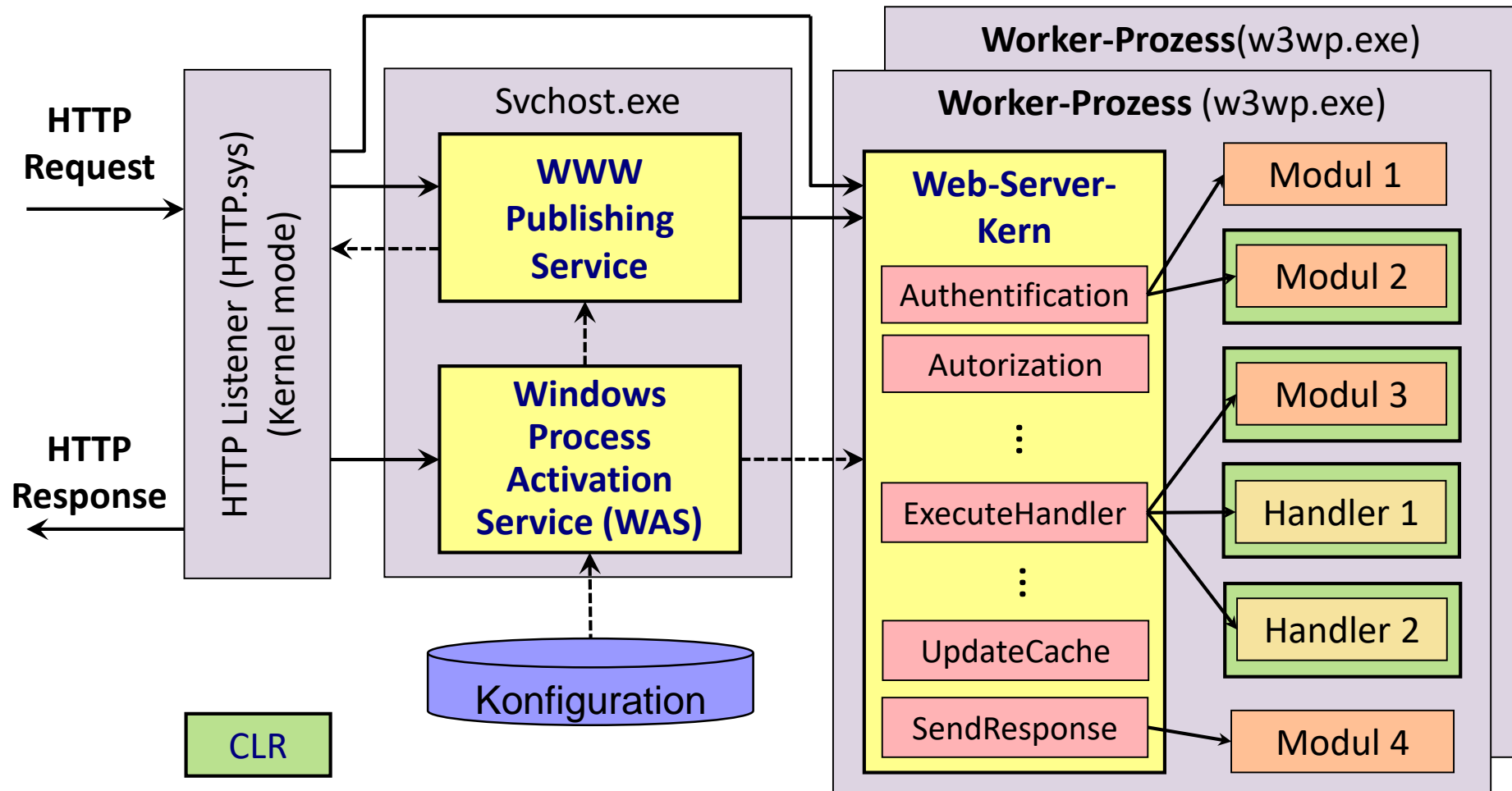
- Grundkonzept und Architektur
- Ein einfaches Beispiel
- Das Routingsystem
- Controller
- Ansichten (Views)
- Dateneingabe
- Validierung
- Testen

ASP.NET

- ASP.NET ist eine Sammlung von Technologien zur Entwicklung von Web-Anwendungen.
- ASP.NET Web Forms
 - Nachfolger von *Active Server Pages (ASP)*, einer der ersten serverseitigen Web-Technologien von Microsoft (1996)
 - Bestandteil des .NET-Frameworks 1 und höheren Versionen (2002)
 - Entwicklung von Web-Seiten auf Basis von Steuerelemente (Web-Controls)
- ASP.NET MVC
 - MVC 1.0 wurde 2009 veröffentlicht
 - Basiert auf dem Model-View-Controller-Muster
- ASP.NET Core
 - Erste Version seit Juni 2016 verfügbar
 - Völlige Neuentwicklung des Web-Stacks auf Basis von .NET-Core
 - Unterstützung verschiedener Betriebssysteme (Windows, OS X, Linux)

IIS 7 und ASP.NET: Systemarchitektur

Eingehende Requests werden auf mehrere Worker aufgeteilt
Application-Pools



ASP.NET Web Forms

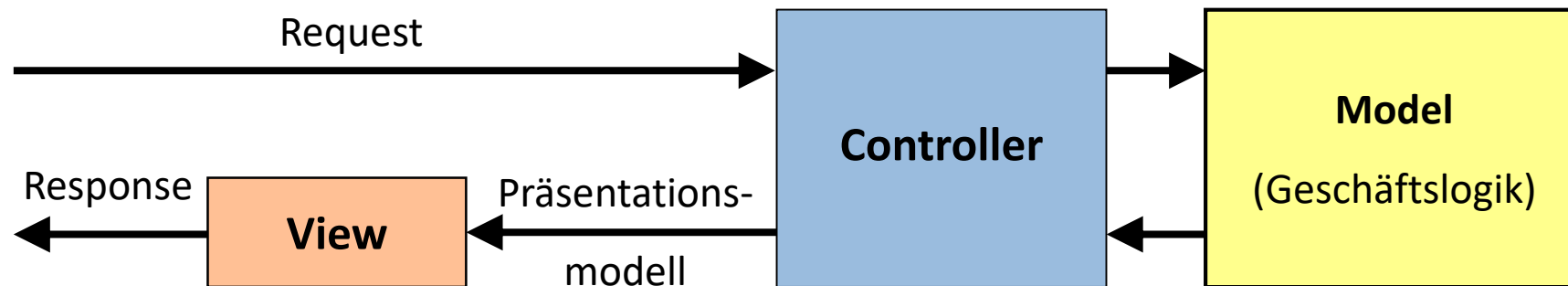
- Ähnliches Programmierkonzept wie bei der Entwicklung von Desktop-Anwendungen (Windows Forms).
- Steuerelemente
 - Aussehen und Verhalten wird durch Eigenschaften bestimmt.
 - Steuerelemente können in HTML-Fragment gerendert werden.
 - Web-Seite wird als Steuerelementebaum repräsentiert.
- Viewstate
 - Steuerelemente haben Zustand, der in Viewstate gespeichert wird.
 - Viewstate wird in Response zum Browser und in Request zurück an den Server geschickt.
- Ereignisgetriebene Programmierung
 - Anwendung reagiert in Callback- und Lebenszyklus-Methoden auf Ereignisse.
 - Änderung von Eigenschaften, Zugriff auf Geschäftslogik.

Probleme von ASP.NET Web-Forms

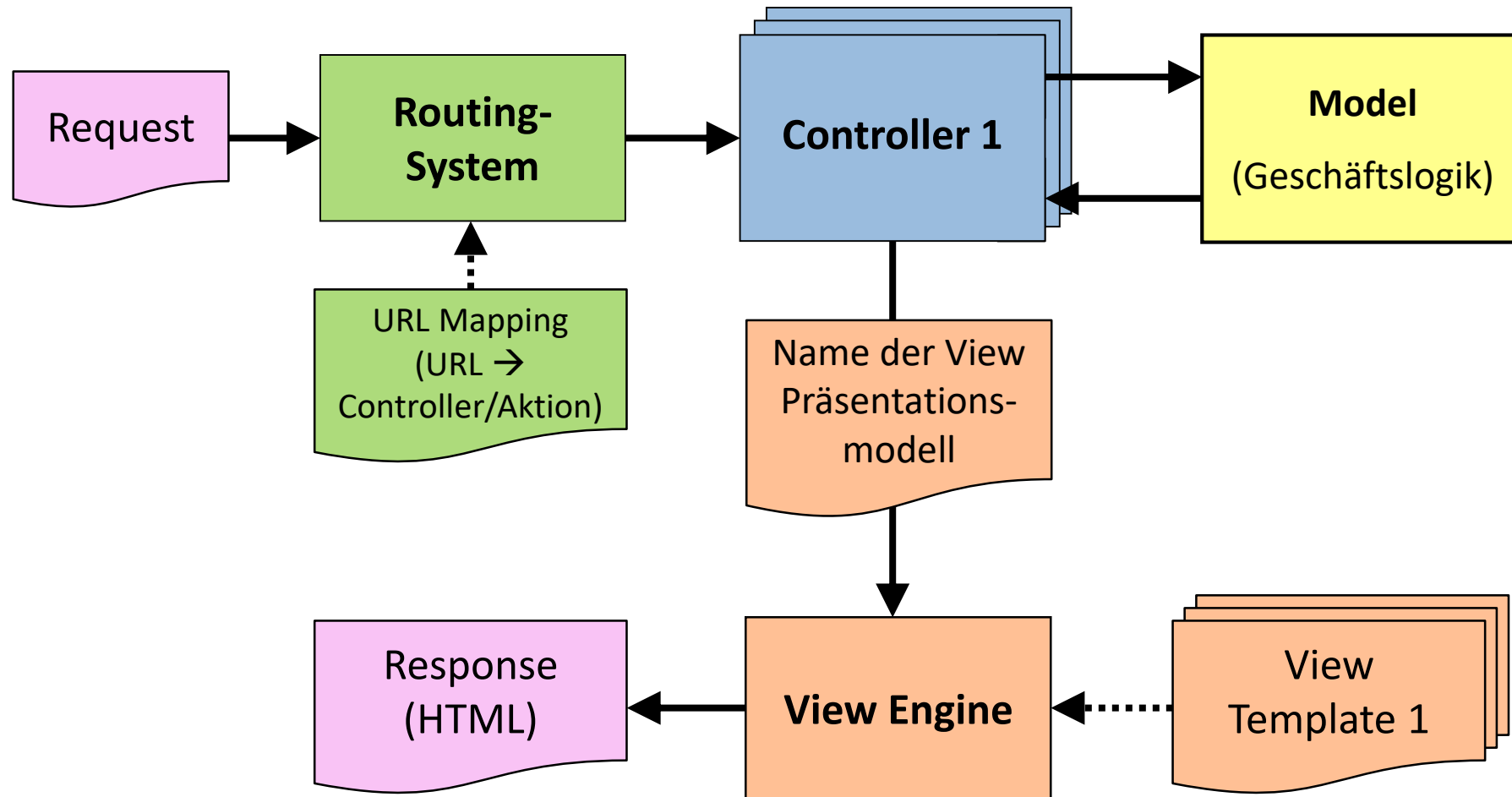
- Laufzeiteinbußen durch Übertragung des ViewStates.
- Komplexer Lebenszyklus einer ASP.NET-Seite.
- Eingeschränkte Kontrolle über HTML
 - Verwendung von CSS teilweise schwierig
 - Probleme mit JavaScript-Code
- Vermischung von Präsentations- und Geschäftslogik im Code-Behind.
- Präsentationslogik kann kaum getestet werden.

ASP.NET MVC – Grundkonzept

- Konsequente Trennung von
 - **Model** (= Geschäftslogik),
 - **View** (= Beschreibung des zu generierenden HTML-Dokuments) und
 - **Controller** (= Präsentationslogik)
- Controller stellt Verbindung zu Geschäftslogik her.
- Controller ist unabhängig von der Ansicht und kann daher einfach getestet werden.



Komponenten von ASP.NET MVC



Eine einfache ASP.NET-MVC-Anwendung (1)

- Implementierung des *Controllers*:

```
public class CurrenciesController : Controller {  
    private ICurrencyCalculator calc = new CurrencyCalculator();  
    public ActionResult Index() {  
        IEnumerable<CurrencyData> model = calc.GetCurrencies().Select(  
                                                    curr => calc.GetData(curr));  
        return View("Index", model);  
    }  
}
```

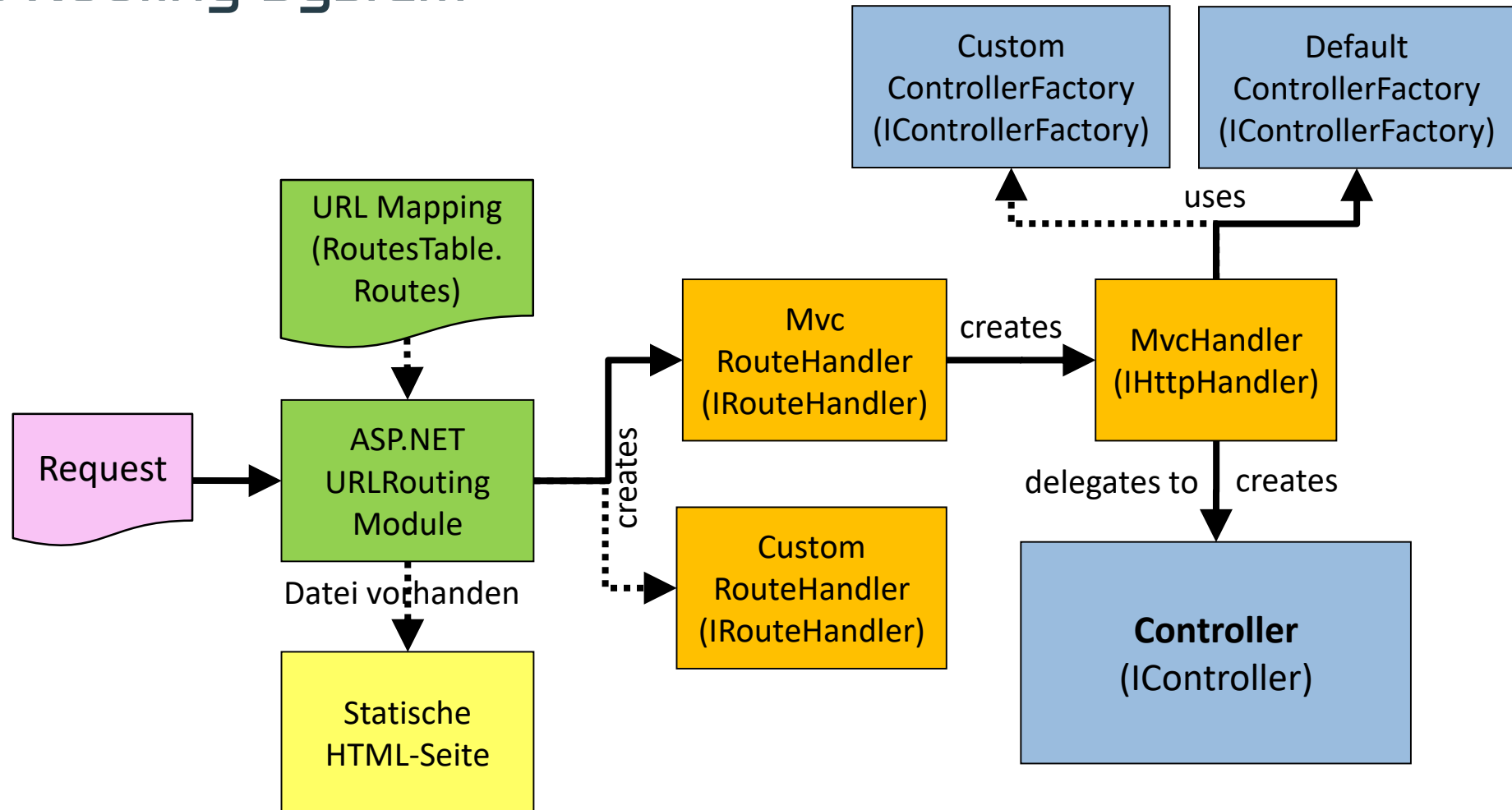
- Das Routing-System weist der URL *http://.../Currencies[/Index]* die Controller-Methode *CurrenciesController.Index* zu.
- Der Controller ermittelt mithilfe der Geschäftslogik (*CurrencyCalculator*) die View-Daten (*model*).
- Der Controller legt auch fest, welches View-Template (*Index.cshtml*) zum Generieren der HTTP-Antwort verwendet werden soll.

Eine einfache ASP.NET-MVC-Anwendung (2)

- Implementierung der *View* (*Views/Currencies/Index.cshtml*):

```
@model IEnumerable<CurrencyData>
<html>
<body>
  <table border="1">
    <tr><th>Code</th><th>Name</th><th>Region</th><th>EuroRate</th></tr>
    @foreach (CurrencyData curr in Model) {
      <tr>
        <td>@curr.Code</td>
        <td>@curr.Name</td>
        <td>@curr.Region</td>
        <td class="rightAligned">@string.Format("{0:F4}",
                                                    curr.EuroRate)</td>
      </tr>
    }
  </table>
</body>
</html>
```

Das Routing-System



Konfiguration des Routing-Systems von ASP.NET

```
protected void Application_Start() { // wird beim Starten der Web-Anw. aufgerufen
    RegisterRoutes(RouteTable.Routes);
}
public static void RegisterRoutes(RouteCollection routes) {
    routes.MapRoute(
        "Default", // Name der Route
        "{controller}/{action}/{id}", // URL mit Parametern
        new { controller = "Home", action = "Index", id = (int?)null } // Standardwerte für Parameter
    );
}
```

URL	abgebildet auf
/	{ controller="Home", action="Index", id=null }
/Products	{ controller="Products", action="Index", id=null }
/Products/List	{ controller="Products", action="List", id=null }
/Products/List/1234	{ controller="Products", action="List", id=1234 }

Routing: Verbindung zu ASP.NET MVC

- *MapRoute* ist eine Erweiterungsmethode, die der Route ein *MvcRouteHandler*-Objekt zuordnet:

```
public static Route MapRoute(this RouteCollection routes, string name,
                             string url, object defaults) {
    routes.Add(name, new Route(url, new MvcRouteHandler())
                { Defaults = new RouteValueDictionary(defaults) });
} ...
```

- *MvcRouteHandler* erzeugt den HTTP-Handler *MvcHandler*.
- *MvcHandler* erzeugt mit Hilfe von *DefaultControllerFactory* einen Controller und ruft die entsprechende *Methode (Action)* auf.
- Beispiel: { controller="Products", action="List", id=1234 }
 - Factory erzeugt ein Objekt der Klasse *ProductsController*.
 - Aufruf der Methode *List()* oder *List(int id)* mit *id=1234*

Controller: Rolle in der MVC-Architektur

- Controller sind der zentrale Einstiegspunkt für jede HTTP-Anfrage.
- An die Controller werden die Anfragedaten in aufbereiteter Form übergeben.
- Controller stellen die Verbindung zur Geschäftslogik her.
- Controller bereiten die Daten für die View vor → Präsentationsmodell
- Controller können getestet werden:
 - Controller liefern strukturierten Ergebnisse (Präsentationsmodell) und keine unstrukturierten HTML-Dokumente.
 - Wegen loser Kopplung kann Geschäftslogik durch Mock-Objekt ersetzt werden.

Controller: Programmiermodell

- Alle Controller implementieren das Interface *Controller*
- Einfachste Form eines Controllers:

```
public class HelloController : System.Web.Mvc.IController {  
    public void Execute(RequestContext req) {  
        HttpResponseBase res = req.HttpContext.Response;  
        res.Write("<h1>Hello World</h1>\n");  
    }  
}
```

- Durch Ableiten von *Controller* werden View und Controller voneinander getrennt:

```
public class HelloController : System.Web.Mvc.Controller {  
    public ActionResult Index() {  
        ViewData["Message"] = "Hello World";  
        return View("Index");  
    }  
}
```

Controller: Eingangs- und Kontextdaten (1)

- Über diverse Behälter (Maps) kann auf Anfrageparameter zugegriffen werden:

Behälter	Beschreibung
Request.{QueryString,Form}	Parameter bei {GET, POST}-Anfrage
Request.Cookies	Cookies der Anfrage
RouteData.Values	Parameter der Route
HttpContext.{Application, Session, Cache}	{globale, sitzungsbezogene, gepufferte} serverseitige Daten
...	

- Beispiel:

```
public ActionResult Convert() {  
    string inValueStr = Request.Form["inputValue"];  
    decimal inValue = inValueStr==null ? 0 : decimal.Parse(inValueStr);  
    string selectedCurrency = Request.Form["selectedCurrency"];  
} ...
```


Controller: Eingangs- und Kontextdaten (2)

- Parameter der HTTP-Anfrage und Routenparameter können auf Controller-Methoden abgebildet werden.
- Konvertierungsfehler werden in *ModelState* gespeichert.

- Beispiel 1: `url/converter?inputValue=100&selectedCurrency=USD`

```
public ActionResult Convert(decimal? inputValue,  
                             string selectedCurrency) {  
    ...  
}
```

- Beispiel 2: `url/persons/list/10000`

```
routes.MapRoute(...,  
    "{controller}/{action}/{id}",  
    new { controller = "Home", action = "Index", id = (int?)null });
```

```
public ActionResult List(int? id, ...) {  
    ...  
}
```

Controller: Response-Objekt

- Das Ergebnis einer Controller-Methode ist eine HTTP-Antwort.
- Die HTTP-Antwort kann mithilfe des Response-Objekts (vom Typ *HttpResponseBase*) direkt generiert werden.

```
public class MyController : Controller {  
    public void Action1() {  
        Response.Cookies["myCookie"].Value = "val";  
        Response.Write("<h1>...</h1>");  
    }  
    public void Action2() {  
        Response.Redirect("/Some/Url");  
    }  
}
```

- Nachteil: Controller- und Präsentationslogik werden vermengt.

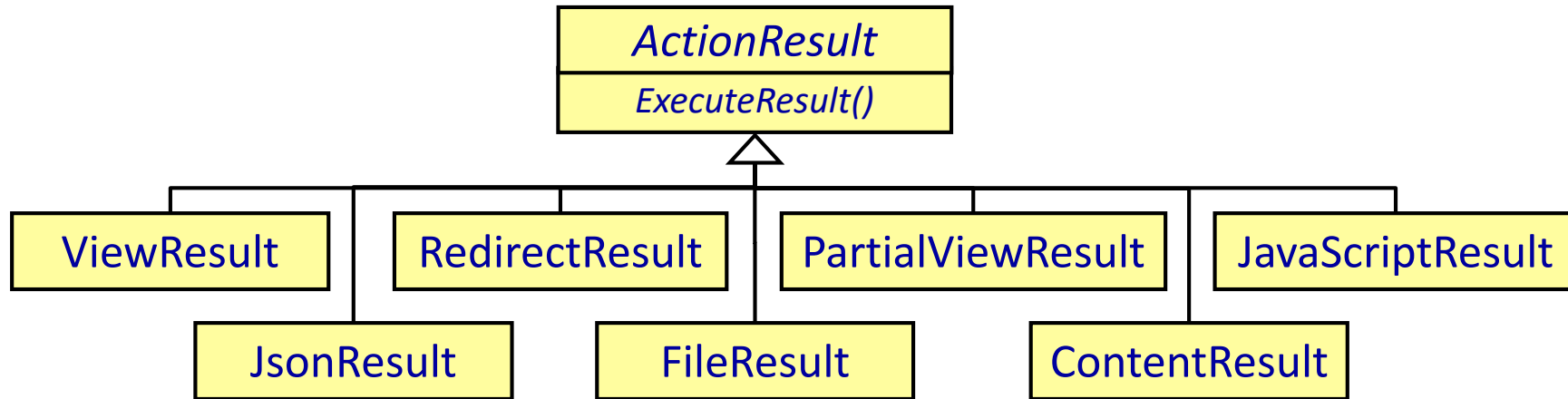
Controller: ViewResult

- In einem *ViewResult*-Objekt werden der Name der View und die View-Parameter (das Präsentationsmodell) an die View-Engine übergeben.

```
public class MyController : Controller {  
    public ViewResult Action1() {  
        ViewData["param1"] = value1;  
        return View("MyView");  
        // oder: return new ViewResult { ViewName = "MyView",  
        //                                     ViewData = this.ViewData };  
    }  
}
```

- Die Standard-View-Engine (*WebFormViewEngine*) sucht an folgenden Stellen nach dem View-Template:
 - Views/ControllerName/ViewName.{aspx,ascx}
 - Views/Shared/ViewName.{aspx,ascx}
 - Views/ControllerName/ViewName.cshtml
 - Views/Shared/ViewName.cshtml

Controller: ActionResult



- Ergebnis einer Controller-Methode ist ein strukturiertes Objekt, das zum Testen verwendet werden kann.

```
public class MyController : Controller {
    public ViewResult Action1() { return View("Index"); }
    public RedirectResult Action2() { return RedirectToAction("Action3"); }
}
```

- HTTP-Antwort wird erst durch Aufruf von *ExecuteResult()* generiert.

Controller: RedirectResult

- In Controller-Methoden möchte man andere Controller-Methoden aufrufen und damit Ansichten gemeinsam nutzen.

```
public ActionResult AddToCart(CartItem item) {  
    businessLogic.AddToCart(item)  
    return this.List();  
}
```

- Problem: URL wird nicht aktualisiert → Probleme bei Aktualisierung der Seite

```
http://host/Cart/AddToCart
```

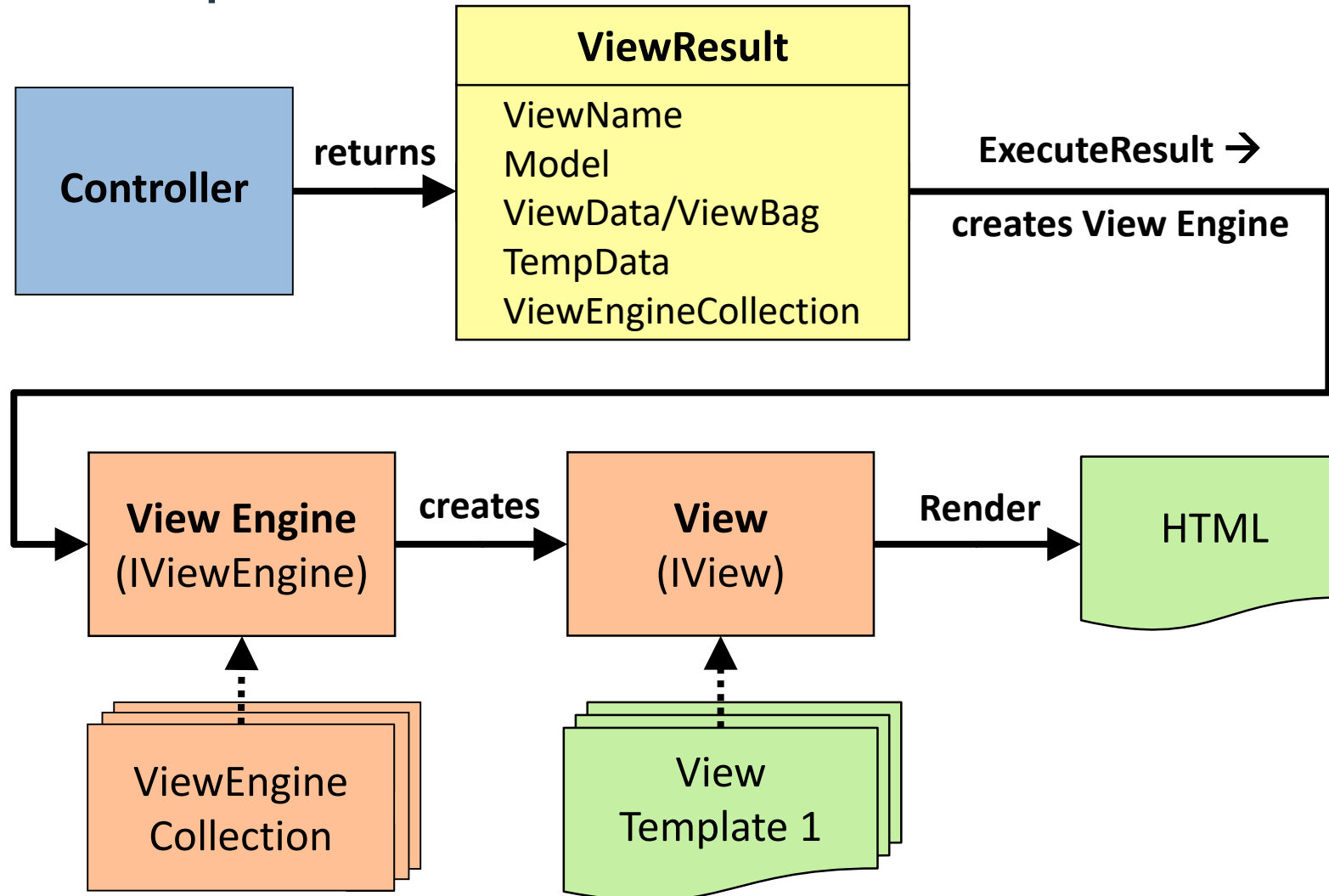
- Soll eine der Ansicht entsprechende URL verwendet werden, ist ein zusätzlicher Roundtrip über den Client (Antwortcode 302) notwendig:

```
http://host/Cart/AddToCart
```

```
public ActionResult AddToCart(CartItem item) {  
    businessLogic.AddToCart(item)  
    return RedirectToAction("List"); // Sendet Antwortcode 302 mit URL http://.../Cart/List  
                                     // Neue Anfrage ruft Controller-Methode List auf.  
}
```

```
http://host/Cart/List
```

Views: Konzept



Views und View-Engines

- ASP.NET MVC unterstützt zwei View-Engines:

WebFormsViewEngine

```
<%@ Page Language="C#" Inherits=
    "ViewPage<IEnumerable<PersonModel>>" %>
<html>
<body>
    <table border="1">
    <tr><th>...</th>...</tr>
    <% foreach (PersonModel p in Model) { %>
        <tr>
            <td><%= p.FirstName %></td>
            <td><%= p.LastName %></td>
            <td><%= p.Email %></td>
        </tr>
    <% } %>
    </table>
</body>
</html>
```

RazorViewEngine (>= ASP.NET MVC 3.0)

```
@model IEnumerable<PersonModel>
<html>
<body>
    <table border="1">
    <tr>...</tr>
    @foreach (PersonModel p in Model) {
        <tr>
            <td>@p.FirstName </td>
            <td>@p.LastName</td>
            <td>@p.Email</td>
        </tr>
    }
    </table>
</body>
</html>
```

- Andere View-Engines können hinzugefügt werden: Spark, NHaml, NVelocity, ...

Views: Das Präsentationsmodell

- Der Controller hat verschiedene Möglichkeiten, Daten an Ansichten zu übergeben:

- Behälter *ViewData*: *Controller*

```
ViewData["message"] = "hello";
```

view

```
<p>@ViewData["message"]</p>
```

- Dynamisch typisierte Datenkomponente *ViewBag*:

```
ViewBag.Message = "hello";
```

```
<p>@ViewBag.Message</p>
```

- Modell-Objekt:

```
public ActionResult Details(  
    int id) {  
    PersonModel model = ...;  
    return View(model);  
}
```

```
@model PersonModel  
...  
<div class="display-field">  
    @Model.FirstName  
</div>
```


View: HTML-Hilfsmethoden

- HTML-Hilfsmethoden erleichtern die Generierung des HTML-Codes.
- Für die meisten Formularelemente existieren Hilfsmethoden, z. B.

```
Html.TextBox("Email",  
    "mayr@gmail.com")
```



```
<input id="Email" name="Email",  
    type="text" value="mayr@gmail.com">  
</input>
```

- Der angezeigte Wert im Formularelement wird folgendermaßen ermittelt:
 1. ModelState["Email"].Value.AttemptedValue, falls Validierungsfehler vorliegt.
 2. Zweiter Parameter der Hilfsmethode, falls vorhanden.
 3. ViewBag.Email
 4. ViewData["Email"]
 5. @Model.Email
- Es existieren typischere Varianten der Hilfsmethoden:

```
Html.TextBoxFor(model => model.Email)
```

View: Selbstdefinierte HTML-Hilfsmethoden

- Möglichkeit 1: Implementierung von Erweiterungsmethoden für die Klasse *HtmlHelper*.
- Möglichkeit 2: Inline-Methoden (bei Verwendung der Razor-View-Engine)

```
@helper CreateList(string[] items) {  
    <ul>  
        @foreach (string item in items) {  
            <li>@item</li>  
        }  
    </ul>  
}
```

```
<p>Person List:</p>  
@CreateList(ViewBag.PeopleNames)
```

- Helper können in das Verzeichnis App_Code ausgelagert werden und stehen so mehreren Views zur Verfügung.

View: Beispiel für HTML-Hilfsmethoden

```
@using(Html.BeginForm()) {  
    @Html.HiddenFor(model => model.Id)  
    <div class="editor-label">  
        @Html.LabelFor(model => model.FirstName)  
    </div>  
    <div class="editor-field">  
        @Html.EditorFor(model => model.FirstName)  
    </div>  
}
```



```
<form action="/Person/Edit/1" method="post">  
    <input id="Id" name="Id" type="hidden" value="1" />  
    <div class="editor-label">  
        <label for="FirstName">FirstName</label>  
    </div>  
    <div class="editor-field">  
        <input id="FirstName" name="FirstName" class="text-box single-line"  
            type="text" value="Franz" />  
    </div>  
</form>
```

Dateneingabe

- Über eine GET-Anfrage wird das HTTP-Formular angefordert:

```
[HttpGet]
public ActionResult Edit(int id) {
    Person pers = persAdmin.FindById(id);
    return View(new PersonModel(pers));
}
```

- Mit einer POST-Anfrage wird das ausgefüllte Formular an den Server gesendet:

```
[HttpPost]
public ActionResult Edit(int id, FormCollection collection) {
    if (ModelState.IsValid) {
        Person pers = new Person { Id = int.Parse(collection["Id"]),
                                   FirstName = collection["FirstName"], ... };
        persAdmin.Update(pers);
        return RedirectToAction("Index");
    }
    else
        return View();
}
```

Dateneingabe: Modellbindung

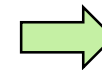
- Formularfelder können auf .NET-Objekt abgebildet werden:

```
[HttpPost]
public ActionResult Edit(
    int id, PersonModel pm) {
    if (ModelState.IsValid) {
        persAdmin.Update(pm.ToPerson());
        return RedirectToAction("Index");
    }
    else
        return View();
}
```

```
public class PersonModel {
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
}
```

- Formularfelder werden auf gleichnamige Eigenschaften abgebildet:

```
<input name="FirstName" type="text" value="Franz" />
```



```
pm.FirstName
```

- Mit dem Attribut *Bind* kann der Abbildungsprozess näher definiert werden:

```
public ActionResult Edit(int id,
    [Bind(Prefix="pers", Exclude="Email")] PersonModel pm) { ... }
```

- Feld mit dem Namen *pers.FirstName* wird auf *pm.FirstName* abgebildet.

Validierung: Validierungslogik

- Validierungsfehler werden im Objekt *ModelState* gespeichert.
 - Eingaben werden vom Framework im Zuge der Modellbindung automatisch auf Korrektheit überprüft.
 - In Contollermethoden können die Eingaben explizit validiert und bei Bedarf Validierungsfehler zu *ModelState* hinzugefügt werden.

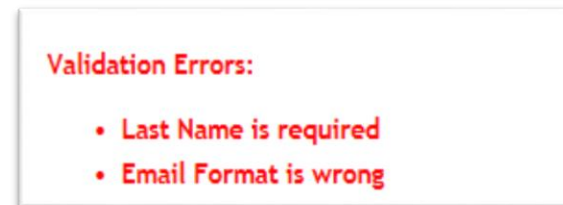
```
public ActionResult Edit(int id, PersonModel pm) {  
    if (ModelState.IsValidField("Email")) // If field Email in correct format ...  
        if (! Logic.IsRegistered(pm.Email)) // If Email is not found in data base ...  
            ModelState.AddModelError("Email", "Unregistered email address");  
  
    if (ModelState.IsValid) { // If all fields are valid ...  
        persAdmin.Update(pm.ToPerson());  
        return RedirectToAction("Index");  
    }  
    else  
        return View();  
}
```

Validierung: Anzeige der Validierungsergebnisse

- HTML-Hilfsmethoden berücksichtigen den Zustand von *ModelState* und generieren entsprechenden HTML-Code.
 - Liegt für ein Eingabefeld ein Validierungsfehler vor, wird in diesem Feld der eingegebene Wert (*ModelState["field-name"].Value.AttemptedValue*) dargestellt.
 - Mit CSS-Regeln (z. B. *.input-validation-error*) kann die Formatierung von fehlerhaften Eingaben festgelegt werden.
- Es existieren HTML-Hilfsmethoden zur Anzeige von Fehlermeldungen:
 - *Html.ValidationMessage("field-name")*:
 - *Html.ValidationSummary()*:



A screenshot of a web form with two input fields. The first field is labeled 'LastName' and is empty, with a red border and the error message 'Last Name is required' to its right. The second field is labeled 'Email' and contains the text 'f.klammer', also with a red border and the error message 'Email Format is wrong' to its right.



A screenshot of a validation summary box. It has a title 'Validation Errors:' in red. Below the title, there are two bullet points in red: '• Last Name is required' and '• Email Format is wrong'.

Validierung: Client-seitige Validierung

- Mit Attributen können Validierungsregeln definiert werden, die bei der client- und serverseitigen Validierung berücksichtigt werden:

```
public class PersonModel {  
    [Required(ErrorMessage="Email is required")]  
    [RegularExpression(@"^\w+([-+.']\w+)*@\w+([-.] \w+)*\.\w+([-.] \w+)*$",  
        ErrorMessage="Email format is wrong")]  
    public string Email { get; set; }  
}
```

- *jQuery*-basierte Validierungsbibliothek muss in HTML inkludiert werden.
- HTML-Hilfsfunktionen reichern HTML-Elemente mit zusätzlichen „data-*“-Attributen an, die zur Validierung verwendet werden:

```
<input data-val="true" data-val-regex="Email format is wrong"  
    data-val-regex-pattern="^\w+..." name="Email" type="text" value="..." />
```

- Konzepte der Validierungslogik
 - Validierungscode ist nicht direkt mit HTML-Elementen verbunden.
 - Falls JavaScript deaktiviert ist, wird serverseitige Validierung durchgeführt (Barrierefreiheit).

Validierung: Beispiel

```
@using (Html.BeginForm()) {  
    <div class="editor-field">  
        @Html.EditorFor(model =>  
            model.Email)  
        @Html.ValidationMessageFor(  
            model => model.Email)  
    </div>  
    @Html.ValidationSummary(true)  
}
```



FirstName

LastName Last Name is required

Email Email Format is wrong

- Last Name is required
- Email Format is wrong

```
<form action="/Person/Edit/1" method="post">  
    <div class="editor-field">  
        <input class="text-box single-line" data-val="true"  
            data-val-regex="Email format is wrong"  
            data-val-regex-pattern="..."  
            id="Email" name="Email" type="text" value="..." />  
        <span class="field-validation-valid" data-valmsg-for="Email" ...</span>  
    </div>  
    <div class="validation-summary-valid" ...><ul>...</ul></div>  
</form>
```

Testen von ASP.NET-MVC-Anwendungen

- Da Controller und Ansichten konsequent getrennt sind, können für Controller sehr einfach Unittests erstellt werden:

```
[TestMethod]  
public void IndexTest() {  
    ConverterController controller = new ConverterController();  
    ViewResult result = controller.Index("USD");  
    ConverterModel model = result.Model as ConverterModel;  
  
    Assert.AreEqual("", result.ViewName);  
    Assert.IsNotNull(model);  
    Assert.IsNotNull(model.CurrencyList);  
    SelectListItem selItem = model.CurrencyList.FirstOrDefault(item => item.Selected);  
    Assert.AreEqual("USD", selItem.Text);  
}
```