WED3 Zusammenfassung 23. August 2017
HSR FS17

I. Introduction of Web

I.1 Benefits

- No need for backup
- Platform independent
- No software update
- Lower investment costs
- Software as a Service

I.2 Liabilities

- No data sovereignity
- Limited calibration possibilities
- Limited/restricted hardware access
- No operation system access
- More expensive deployment strategies

II. What is Routing?

- Links multiple application parts together
- Provides the concept of information architecture (IA)

II.1 Basics

- Routing is accomplished **completely** client-side
  - No page reload, no roundtrip, server isn't involved
  - Page transition is managed by JS completely
  - Working back-button and bookmarks
- Entry Point [View UI controller] is enforced by the given route
  - Controller provides features behind a View (UI) and bootstraps it
- Router provides client-side event hooks during navigation - Lifecycle management

II.2 Client-side routing concepts

- **The old way** - Earlier, we used anchors (#). Don't use these anymore!
- **The HTML5 way**
  - JavaScript API window.history is used
  - window.history.pushState causes the address bar to show the URL, but won't cause the browser to load it (or even check, if it's valid)
  - window.onpopstate can be used to listen for route changes
  - **warning:** configuration adjustments needed on server-side (all sub-routes must return root-files)

II.3 Example routeConfig

Listing 1: routeConfig.js

```
let router = new ui.Router({
    rootPath: "/demo3",
    initialRoute: "index",
    routes: {
        "index": () => { controller.indexAction(routerOutletView); }
    }
});
```

III. Data Bindings

```
<p>Your team is {{counter.team}}</p>
<p>Your current count is: {{counter.count}}</p>
<form>
    <button data-click="up">Count Up</button>
</form>
```

```
class CounterModel {
    constructor(team, count) {
        this.team = team || "unspec";
        this.count = count || 0;
    }
}
```

IV. Services

- ...contain the major application logic
- ...are generally the source of all application data **Data Services**
- Provide microtesting of smalles possible logic units
- Completely decoupled from UI
- **UI Services** are usually seen in the communication between UI controllers.

```
class CounterController {
    constructor(counterService) {
        this.counterService = counterService;
    }

    indexAction(viewRef) {
        // The service returns the model and the
        // view is rendered using the returned model.
        this.counterService.load((model) => {
            this.renderIndexView(viewRef, model);
        });
        $(viewRef).on('click', '[data-click=up]', (e) => {
            this.counterService.up(model) => {
                this.renderIndexView(viewRef, model);
            }
            e.preventDefault();
        });
    }
}
```

V. Bundling SPAs

- All JS code must be delivered to the client over potentially metered/slow networks
- Bundling and minifying the source leads to smaller SPA footprint
- Larger SPAs with many modules need a reliable dependency management
- Initial footprint can be reduced by loading dependent modules on-demand

V.1 WebPack as bundler

- **Entry** - The entry point (modules to be bundled) tells webpack where to start and follows the graph of dependencies to know what to bundle.
- **Output** - Tell webpack where to bundle your application
- **Loaders** - Loaders in webpack transform these files into modules as they are added to your dependency graph.
- **Plugins** - Loaders only execute transformations on a per-file basis, plugins are most commonly used performing actions and custom functionality.

Listing 2: webpack.js

```
context: rootDir,
entry: {
    di: srcDir + scriptsDir + "/di.js",
    ui: srcDir + scriptsDir + "/ui.js"
}, output: {
    path: distDir + scriptsDir, filename: "[name].js"
}, module: {
    loaders: [ { test: /\.js$/,
        exclude: /(node_modules|tmp)/,
        loader: 'babel-loader' }]
}, plugins: [
    new HtmlWebpackPlugin({
        title: 'index',
        filename: 'index.html', // Rel. path from "output" dir
        template: srcDir + '/index.html' // Src file
    }),
    new webpack.optimize.UglifyJsPlugin({
        compress: { warnings: false }
})]
```

VI. Angular 2

VI.1 Angular CLI

```
npm install -g @angular/cli // Install the CLI globally
ng new my-app // Create a new angular app
ng serve --open // Serve the Angular app and open the browser
ng build // Just build the angular app
ng test // Build the angular app and execute the test runner
ng generate module core
```

VI.2 Architectural Overview

- **Modules** - A cohesive block of code dedicated to closely related set of capabilities.
- **Directives** - Provides instructions to transform the DOM.
- **Components** - A component is a directive-with-a-template; it controls a section of the view.
- **Templates** - A template is a form of HTML that tells Angular how to render the component.
- **Metadata** - Describes a class and tells Angular how to process it.
- **Services** - Provides logic of any value, function or feature that your application needs.

VI.3 About Modules

- Every app has at least one Angular module (the **root** Module)
- Modules export features (directives, services, ...) required by other modules
- NICHT zu verwechseln mit ES6 Modules (ES6=pro file; Angular=logischer Block von mehreren ES6 Modulen)
- Library Modules:
  - May accomodate multiple Angular modules
  - Contain and export also other facilities (classes, functions, ...)
  - Angular ships as multiple library modules (all with the @angular-prefix)
  - As an ES6 module, the module library provides single export with all containing features (also known as **barrel** export)

VI.4 Modules

- **Root Module** - By convention named AppModule (app.module.ts). Provides the main view, called the root component, that hosts all other app views. Is bootstrapped by the main.ts
- **Core Module** - Provides globally required services and components directly needed by the root module. The core module should help keep the Root Module clean. Only the root Module should import the Core Module.
- **Shared Module** - Provides globally used components/directives/pipes. It's a global UI component module. Do not specify app-wide singleton providers (services) in a shared module (use Root Module instead).
- **Feature Module** - Splits the application into cohesive feature sets. Allows to assign development responsibilities to different teams. Feature modules are designed to extend the app. A feature module can expose or hide it's implementation from other modules.
- **Lazy Module** - Provides similar features such as Feature Modules. Reduces initial footprint of your SPA. Lazy loaded when invoked by a lazy route. Has it's own DI Container (a child of the root injector).

Listing 3: app.module.ts

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {FormsModule} from '@angular/forms';
import {HttpModule} from '@angular/http';
import {CoreModule} from '../core/core.module';
import {AppComponent} from './app.component';
import {AppRoutingModule} from './app-routing.module';
import {AuthModule} from './auth/auth.module';
import {NgbModule} from 'dag-bootstrap/ng-bootstrap';
import {DashboardModule} from './dashboard/dashboard.module';
import {DashboardRoutingModule} from './dashboard/dashboard-rou

@NgModule({
  declarations: [ AppComponent ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    AppRoutingModule,
    DashboardRoutingModule,

    NgbModule.forRoot(),
    CoreModule.forRoot(),
    AuthModule.forRoot(),
    DashboardModule.forRoot(),

    AppRoutingModule
  ],
  providers: [ ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Listing 4: dashboard.module.ts

```
import {NgModule, ModuleWithProviders} from '@angular/core';

import {AuthService} from '../auth/services/auth.service';
import {DashboardComponent} from './components/dashboard/dashboard
import {DashboardRoutingModule} from './dashboard-routing.module
import {RouterModule} from '@angular/router';

@NgModule({
  // declarations (Components / Directives) used
  // from/visible to Module
  declarations: [ DashboardComponent ],
  // Other Modules to import (imports the exported
  // Components/Directives from the other module)
  imports: [ DashboardRoutingModule, RouterModule ],
  // components/Directives (or even Modules)
  // to export (available for other modules; and forRoot() )
  exports: [ ],
  // DI Providers (Services, Tokens, Factories ...),
  // may be instantiated multiple times
  providers: [ AuthService ]
})
export class DashboardModule {
  static forRoot(config?:{}):ModuleWithProviders {
    return {
      ngModule: DashboardModule,
      providers: [ ]
    };
}}
```

VI.5 @NgModule() Metadata

- `declarations[Type1, Type2, ...]` - The *view classes* that belong to this module. Angular has 3 view classes: components, directives and pipes.
- `exports[Type1, Type2, Module1, Module2, ...]` - The subset of declarations that should be visible and usable in the component templates of other modules. Can re-export other modules, which are automatically included when importing this module.
- `imports[Module1, Module2, ...]` - Specifies the modules which exports/providers should be imported into *this* module.
- `providers[Provider1, Provider2, ...]` - Creators of services that this module contributes to the global collection of services (Dependency injection container); they become accessible in all parts of the app.
- `bootstrap[Component]` - The main application view, called the *root component*. Only the *root module* should set this property (enables usage of the root HTML tag: `<app-root>`).

VI.6 Module metadata and provider accumulation mechanisms

- Default import - Imports all components, Pipes, Directives from the given ForeignModule. *Declarations will be re-instantiated on the current module level.* Providers are registered into the current DI container, if registration not yet made.
- `forChild(config?)` import - Represents a static method on a module class (by convention). It is nearly the same as a default import, but allows you to configure services for the current Module level. It returns an object with a `providers` property and an `ngModule` property.
- `forRoot()` `import` - Represents a static method on a module (by convention, see `forChild()` import). This type of import is useful when you want to enforce that the same provider **won't** be loaded twice by lazy modules.
  - **Only root modules** should import foreign Modules by calling `forRoot()`
  - Declare your providers in `@NgModule` declaration OR in `forRoot()`, but **never** in both.
  - The `providers` are added to the DI container on **root** level
  - Also, the other ForeignModule are imported by the `NgModule` property.
  - Providers from `ForeignModule.forRoot()` take precedence over the `providers` from the module definition.

VI.7 Components

Components control and support the view (Controller in MVC / ViewModel in MVVM). Declared as a TS class with an `@Component` function decorator. The lifecycle is managed by Angular (Hydration, Update, Dehydration)

Listing 5: payment.component.ts

```
import {Component, OnInit} from '@angular/core';
import {NgForm} from '@angular/forms';
import {AuthService} from '../../../auth/services/auth.service'
import {AccountsService} from '../../../services/accounts.service'

@Component({
  selector: 'app-payment',
  templateUrl: './payment.component.html',
  styleUrls: ['../payment.component.css']
})
export class PaymentComponent implements OnInit {
  @Output() click = new EventEmitter<any>();
  @Input() title: string;
  private sender: AccountViewModel;
  private recipient: AccountViewModel = new AccountViewModel();
  private amount: number = 0;

  constructor(private authSvc: AuthService,
              private accSvc: AccountsService) { }

  ngOnInit() {
    this.sender = new AccountViewModel(this.authSvc.authenticatedUser);
  }

  public recipientChanged(event) {
      this.accSvc.fetchAccountOwner(this.recipient.accountNr)
        .subscribe((nr) => { this.recipient.nr = nr; });
  }
}
```

VI.8 Templates

- Almost all HTML syntax is valid template syntax (except `<script>` for security reasons). Some legal HTML doesn't make much sense in a template (`<head>`, `<body>`)
- Angular extends the HTML vocabulary of your templates with:
  - Interpolation
  - Template Expression/Statements
  - Binding Syntax
  - Directives
  - Template Reference Variables
  - Template Expressoin Operators (Advanced)
- Binding Syntax
  - Two Way Binding [()]: `<input type="text"[(ngModel)]="counter.team">`
  - One Way (View to Model / Event Binding) (...): `<button (click)="counter.eventHandler">`
  - One Way (Model to View / Property Binding) [...] or {{ ...}}: `<input type="text"[(ngModel)]="counter.team">`
- Binding to targets must be declared as Inputs or Outputs (like in the example above)

VI.9 Directives

Similar to a component, but without a template. Declares as a Typescript class with an `@Directive()` function decorator. Two different kind of directives exist: Strucutral directives (Modifies the structure of your DOM) and Attribute directives (Alter the appearance or behavior of an existing element)

- Attribute Directive `<div [style.font-size]=isSpecial ? 'x-large' : 'smaller'>`
  - **NgStyle Directive** `<div [style.font-size]=isSpecial ? 'x-large' : 'smaller'>`
  - **NgClass Directive** `<div [class.special]=isSpecial>`
- Structural Directives
  - Asterisk is $syntactic sugar" for something a bit more complicated
  - Angular desugars in two stages: First it translated the `*directive="..."` into a template *attribute*, `template="directive ..."`. Then it translates the attribute into a `<template>` Element.
  - Example: `<div *ngIf="hasTitle">` results in `<template [ngIf]="hasTitle"><div>`

Template reference variables

- References a DOM element within a template
- Can also be a reference to an Angular component or directive
- Reference variables can be used anywhere in the template
- A hash symbol (#) declares a reference variable
- Example:

```
<input placeholder="phone_number" #phone>
<!-- phone refers to the input element -->
<button (click)="callPhone(phone.value)">Call</button>
```

VI.10 Services

- Provides any value, function, or feature that your application needs.
- Almost anything can be a service - it should do one thing and do it well.
- Typical services are logging service, data service, message bus, tax calculator, application configuration
- Strongly coupled to Dependency Injection (Angular uses DI to provide the services to the components who need them. Therefore services must be registered in teh DI Container)

Use the `@Injectable` decorator for services.

```
@Injectable()
export class CounterService { }
```

Then you need to register the service within the DI contianer

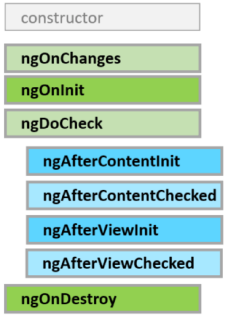
```
@NgModule({
  imports: [...],
  declarations: [...],
  providers: [ CounterService ],
  exports: [...]
})
export class CounterModule { }
```

To use the Service in a component, you can declare it in the constructor and it will be injected by the DI Container.

```
@Component(...)
export class CounterComponent {
  private counter : CounterModel;
  constructor(private counterService: CounterService) {
    this.counter = counterService.load();
  }
}
```

VI.11 Component Lifecycle

Green events are more important



- `ngOnInit` - the creation event (also known as **hydration**) Setup the component and initially fetch data from an underlying data source (do not put too much logic here, just load data and delegate to other methods)

- `ngOnDestroy` - the destruction even (also known as **dehydration**) Use this method to detach event handlers to avoid memory leaks.

```
@Component( { ... })
export class CounterComponent implements OnInit, OnDestroy {
    ngOnInit() { console.log('OnInit'); }
    ngOnDestroy() {console.log('OnDestroy'); }
}
```

VI.12 Component Transclusion

Angular components consist of a view (HTML) and the component logic (Class). Reusable angular components enable parameterization of the view. Transclusion allows the component user to add content to the body section.

```
<section>
  <wed-navigation>
    <h1 wed-title>WED3 Lecture</h1>
    <menu><!-- ... --></menu>
  </wed-navigation>
</section>
```

```
<header>
  <ng-content select='[wed-title]'></ng-content>
</header>
<nav>
  <ng-content select='menu'><ng-content>
</nav>
```

VI.13 Asynchronous Services

In Angular, you can use RxJS or EventEmitters to handle async requests / responses. We'll focus on EventEmitters, where you have to subscribe to an event.

```
@Injectables()
export class SampleService {
  public samplesChanged:EventEmitter<SampleModel[]> =
    new EventEmitter<SampleModel[]>();
  load(): void {
    /* In real world, invoke data resource service here */
    this.sampleChanged.emit(this.samples);
  }
}
```

Receiving the data

```
@Component({...})
export class SampleComponent implements OnInit, OnDestroy {
  ngOnInit() {
    this.samplesSubscription = this.samplesService
      .samplesChanged.subscribe(
        (data:SamplaModel[]) => { this.samples = data; }
      )
  }
  ngOnDestroy() {
    this.sampleSubscription.unsubscribe();
  }
}
```

VI.14 HTTP Client API with Observables

About Observables Think of an observable as a Stream: To listen to objects in the stream, subscribe to the observable. There are Hot Observables and Cold Observables. Hot Observables are shared among all subscribers (for sequences of events, such as mouse move or stock tickers). Cold Observables start running on subscription (such as async web requests) and are not shared among subscribers. They are automatically closed after the task is finished (as opposed to Hot Observables, which do not close automatically).

Angular HTTP API is implemented as a Cold Observable, therefore each subscription will result in a new HTTP Request. The `subscribe()` method listens for events of an Observable. This method consumes three function pointers:

- onNext - defines, what's to-do when data becomes available.
- onError - an error has been thrown while processing the observable. Depending on the implementation, the stream might be broken.
- onComplete - The task has been completed. The stream is about to be closed.

```
var subscription = this.http.get('api/samples').subscribe(
  function (x) { /* onNext -> data received (in x) */ },
  function (e) { /* onError -> the error (e) was thrown */ },
  function () { /* onCompleted -> the stream is closing down */ }
);
```

```
@Injectable()
export class SampleDataResourceService {
    constructor(private http: Http) { }

    get(): Observable<SampleModel[]> {
        return this.http.get('/api/samples')
            .map(this.extractData)
            .catch(this.handleError);
    }
    private extractData(res: Response) {
        let body = res.json();
        return body.data || {};
    }

    private handleError(error: Response | any) {
        return Observable.throw(error.message);
    }
}
```

VI.15 Angular Routing

Use Angular Router to navigate among views. Once the application is bootstrapped, the Router performs the initial navigation based on the current browser URL. Angular Router is an external Module called RouterModule. It's important to add `<base href>` to the index.html site.

Defining the Router Outlet RouterOutlet is a directive from the router library. It defines where the router should display the views. Can also be specified within a child component.

```
<h1>WED3 - App Component</h1>
<nav>
    <a routerLink="/welcome">Welcome Page</a>
</nav>
<router-outlet></router-outlet>
```

Listing 6: example-routing.module.ts

```
const appRoutes: Routes = [
    {     path: '',
          component: DashboardComponent,
          canActivate: [AuthGuard],
          children: [{
              path: '', canActivateChild: [AuthGuard],
              children: [
              {path:'',component:OverviewComponent},
              {path:'about',component:AboutComponent},
              {path:'**',component:NotFoundComponent}]
    }]}
];
```

VI.16 Angular Forms

There are template driven and reactive (model-driven) forms. We focus on template driven forms. By using the `<form>` tag, Angular automatically replaces it with an `ngForm`. It provides additional validation and error handling features. Use standard HTML5 features to validate your form. Use the `[(ngModel)]` binding to bind values. This reads out the value of the model for the first time. Updates are automatically written back into the bound model.

```
<form (ngSubmit)="doLogin(frm)" #frm="ngForm">
    <input type="text" class="form-control" id="login" required
      [(ngModel)]="model.login" name="login" #name="ngModel">
    <div [hidden]="name.valid || name.pristine" class="alert">
        Name is required!
    </div>
    <button type="submit" [disabled]="!frm.form.valid" class="btn
        Submit
    </button>
</form>
```

VII. React

React ist eine Library (kein Framework!) um UI's zu bauen. Es besitzt ein minimales Featureset und wurde vom Gesichterbuch entwickelt.

Prinzipien von React Functionale Programmierung: Komponenten sind Funktionen von `(Attribute, State?) => View`. Komposition statt Vererbung. Immutability. Minimieren von und expliziter mutable State. Braucht es einen State/Lifecycle? Dann verwende eine Klassenkomponente. Sonst verwende lediglich eine Funktion (`function Hello(props)`).

VII.1 JavaScript XML (JSX)

React verwendet JSX, einen Präprozessor, der JavaScript um XML ergänzt – XML kann an beliebiger Stelle vorkommen.

JSX Einschränkungen

- React Elemente müssen mit Grossbuchstaben anfangen. JavaScript-Keywords dürfen nicht verwendet werden.
- React muss immer importiert werden, wenn JSX verwendet wird. Weil JSX vom Präprozessor zu `React.createElement` Aufrufen umgewandelt wird.

VII.2 Props and State

Komponenten erhalten alle Parameter als `props` Objekt (bei Klasse als `this.props` und bei Funktionen als Parameter). **Props sind immer read-only**. React **Klassenkomponenten** können einen veränderbaren Zustand haben. Um den State zu ändern, verwenden wir die Methode `setState()`. Ist der nächste State vom Vorherigen abhängig, sollte man diese folgende Form verwenden (falls der neue State unabhängig vom alten ist, kann `state =>` weggelassen werden).

```
class Counter extends React.Component {
    state = {
        counter: 0
    }

    increment() {
        this.setState(state => ( {
            counter: this.state.counter + 1}));
    }

    render = () => {
        <div>
            {this.state.counter}
            <button
                onClick={this.increment.bind(this)}>Add</button>
        </div>
    }
}
```

VII.3 React CLI

```
npm install -g create-react-app
create-react-app hello-hsr
npm start       // (Starts the development server)
npm run build   // (Bundles the app into static files for producti
npm test        // (Starts the test runner)
npm run eject   //(Removes this tool and copies build dependencies
                // config files, scripts into the app directory.
                // If you do this, you can't go back!)
```

VII.4 React Lifecycle

- Mounting
  1. `constructor(props)` - State initialisieren
  2. `render()`
  3. `componentDidMount()` - DOM aufgebaut, Remote Daten laden, setState führt zu Re-Rendering
- Updating
  1. `componentWillReceiveProps(nextProps)` - Vorschau auf die nächsten Props.
  2. `shouldComponentUpdate(nextProps, nextState)` - wenn `return false`, wird Rendering übersprungen.
  3. `componentWillUpdate(nextProps, nextState)` - selten gebraucht (evtl. Animationen starten)
  4. `componentDidUpdate(prevProps, prevState)` - DOM ist aktualisiert.
- Unmounting
  1. `componentWillUnmount()` - Aufräumen

VII.5 Container vs Presentation Component

Trenne die Präsentation von der Logik. Anstatt eine Komponente zu bauen, die sowohl den Lifecycle und die Rechenarbeit macht, wie auch die Daten darstellt, baue zwei Komponenten. Meistens ist die Präsentationskomponente eine reine Funktion und die Container Komponente eine Klasse.

VIII. Redux

Bei grösseren Anwendungen kommt oft Redux (Predictable State Container) zum Einsatz. Der State wird als Tree von Objekten dargestellt. Ein Tree für die gesamte Applikation! Alle Veränderungen am Tree führen zu einem neuen Tree (immutable). State wird im sogenannten **Store** verwaltet.

IX. ASP.NET (Core)

ASP.NET ist eine der am weitesten verbreiteten Technologien für das Erstellen von Websites.

IX.1 Grundlagen

Attribute Attribute werden verwendet, um die Konventionen von ASP zu überschreiben oder zu unterstützen.

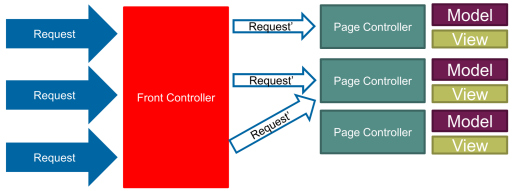
```
public class Test {
    [Required]
    [StringLength(100, MinimumLength=10)]
    public string Name {get;set;}

    [HttpPost]
    public ActionResult About();
}
```

Multithreading

- ASP.NET besitzt einen Threadpool (grösse konfigurierbar)
- ASP.NET wählt für jeden Request einen Thread aus dem Pool. Dieser bearbeitet die Anfrage.
- Der Thread ist so lange blockiert, bis der Request abgeschlossen ist. Es gibt aber Möglichkeiten, den Thread frühzeitig zurückzugeben.
- Warnung: Keine geteilten Daten in Controller und Service halten (z.B. statische Variablen). **ASP instanziiert für jeden Request einen neuen Controller.**

Front Controller In ASP.NET übernimmt der Front Controller das Routing.



IX.2 Middlewares

Ein Request durchläuft ein Stack von Middlewares. **Jede Middleware kann den Request beenden.** Beispiele für Middlewares: Autorisierung, Logging, Welcome Page, Static Files ASP.NET kennt 4 verschiedene Varianten, um Middlewares zu registrieren (die 4. ist die Middleware als Klasse).

Listing 7: Middleware registration example

```
// Registriert neue Middleware
app.Use(async (context, next) => {
    System.Diagnostics.Debug
      .WriteLine("Handling request");
    await next.Invoke();
    System.Diagnostics.Debug
      .WriteLine("Finished handling request");
});
// Erzeugt Verzweigung fuer den angegebenen Anfragepfad.
app.Map("/logging", builder => {
    builder.Run(async (context) => {
        await context.Response.WriteAsync("Hello World");
    })
});
// Terminiert den Request, keine
// weitere Middlewares werden aufgerufen.
app.Run(async (context) => {
    await context.Response.WriteAsync("Hello World");
});
```

IX.3 Dependency Injection - Registration

Wenn als Parameter (sowohl im Konstruktor oder auch im Request Handler eines Controllers) ein Interface erwartet wird, wird im DI Container nachgeschaut ob es eine Dependency zum Injecten gibt. Eine Captive Dependency ist eine Dependency mit falsch konfigurierter Lifetime (z.B. sie wird gar nie verwendet).

Listing 8: DI Registration example

```
public class Startup {
    // This method gets called by the runtime.
    // Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services) {
        services.AddTransient<IUserService, UserService>();
        //services.AddTransient<IUserService, FakeUserService>();
    }

    // This method gets called by teh runtime.
    // Use this method to configure the HTTP Request pipeline
    public Configure(IApplicationBuilder app,
        IHostingEnvironment env,
        ILoggerFactory loggerFactory) {
            app.UseMiddleware<UserMiddleware>();
    }
}
```

Dependency Lifetime

- **Transient** - are created each time they are requested. This lifetime works best for lightweight, stateless services.
- **Scoped** - are created once per request.
- **Singleton** - are created the first time they are requested (or when ConfigureServices is run if you specify an instance there) and then every subsequent request will use the same instance.

Wichtig: Multi-Threading beachten (z.B. `DBContext` ist nicht Thread-Safe).
Merke: Komponenten dürfen sich nur Komponenten mit gleicher oder längerer Lebensdauer injecten lassen.

IX.4 Controller

Der Controller beinhaltet die Actions, welche vom Framework aufgerufen werden. Parameter vom Query String und Body werden automatisch auf die Method-Parameter von der Action gemapped. Der Controller wird in der Default-Konfiguration für jeden Request neu erzeugt.
Konvention: Postfix "Controller", z.B. "Home**Controller**"
Als Return Value wird ein `ActionResult` Objekt zurückgegeben. Dieses Resultat wird dann zum Client zurückgeschickt.

URL Pattern URL: `http://localhost:5000/{controller}/{action}/`

- {**controller**} Sucht im Folder Controllers nach einer Klasse mit {Name}Controller
- {**action**} Sucht innerhalb dieser Klasse nach einer Methode mit {Name}

```
app.UseMvc(routes => {
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id:int?}");
    )
    routes.MapRoute(
        name: "default2",
        template: "{controller}/{action}/{id?}",
        default: new {controller="Home", action="Index"},
        constraints: new {id=new IntRouteConstraint()});
});
```