Angular Advanced

Getting started

- Create directory c:\course
- Open command prompt and cd into c:\course
  - git clone https://github.com/tutorit/ang240916 trainer
  - git clone https://github.com/tutorit/ang240916 mywork
- Cd into mywork\BookApp
  - npm i –g @angular/cli
  - npm i
  - ng serve --open
- Cd into mywork\server
  - npm i
  - node server.js

- Material is available at (either) cloned directory material-subfolder

- When we start to work with exercises only modify code in mywork directory
- The instructor will occasionally push his examples to the same repository
  - You can git pull at trainer directory to get the latest examples

Topics

Basics Reviewed

- Components and Databinding
- Dependency Injection

Routing between views

- Router-configuration
- Url-strategies
- Route parameters
- Child routing
- Route watches

Modules and Libraries

- Designing modules
- Implementing modules
- Lazy loading
- Implementing libraries

RxJS

- Working with Observables

State management

- Application state vs Component state
- State management libraries

Internationalization

- Data formatting
- Translations

Progressive Web Applications

- What is PWA
- Service Workers

Server side rendering

- Why and how
- Angular Universal

Other topics

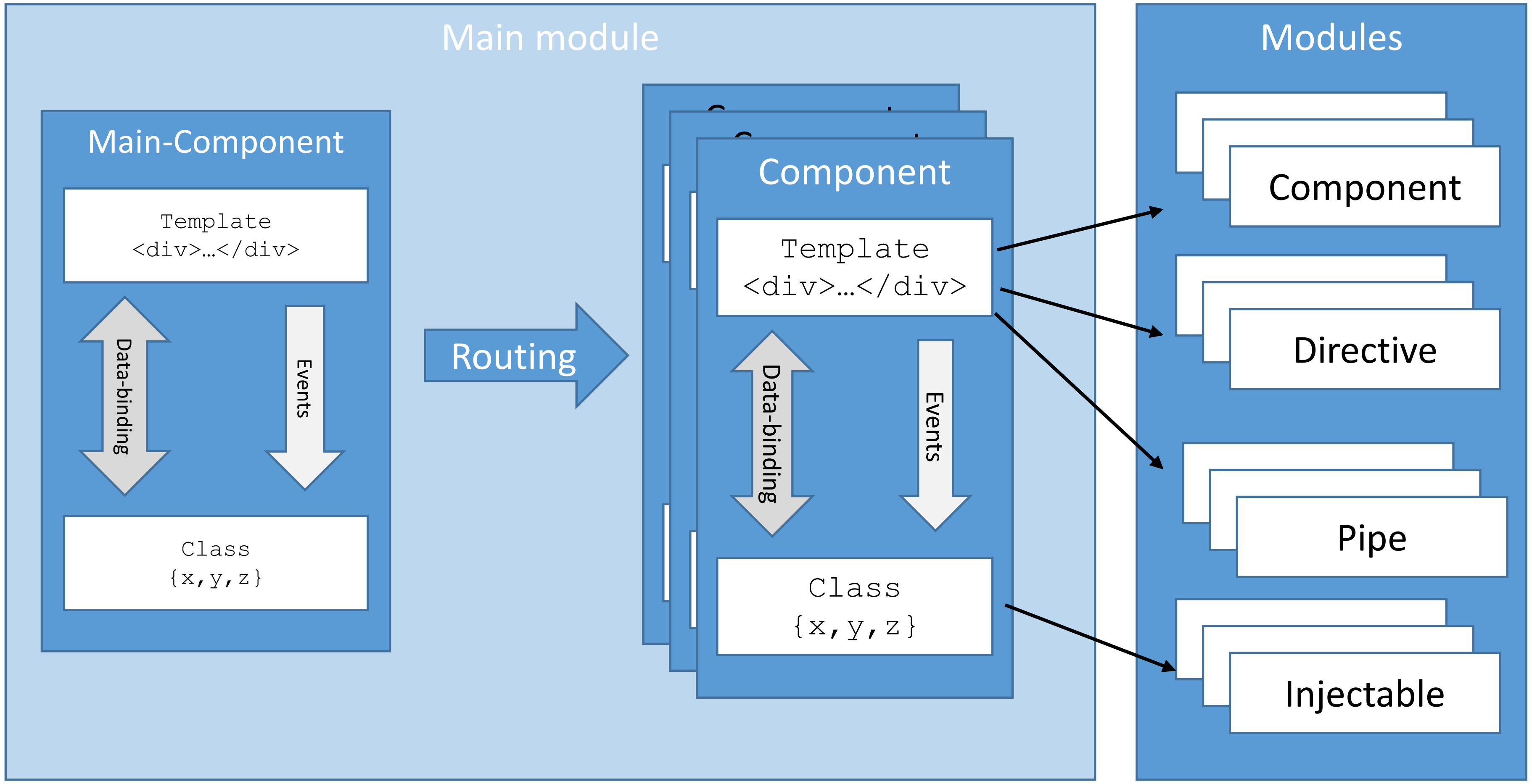
- Web Workers
- Security

Basics Reviewed

Components and Databinding

Services and Dependency Injection

Angular Architecture



Angular cli

Angular cli is the preferred command line tool for implementing angular applications

- Creates project
  - Strict about directory structure
- Code generator
  - Cli commands help implementing basic constructs
  - Componenents
  - Services
  - Routing and routes
- Compiles
  - Compiles typescript
  - Includes AOT-compiler for Angular constructs

```
➢ npm install -g @angular/cli
➢ ng new BookApp --no-standalone
➢ cd BookApp
➢ ng serve --open
```

Angular cli-project

Project directory

- Configuration files
- src-directory
  - index.html
  - styles.css
  - main.ts
  - app-directory
    - app.module.ts
    - app.component.ts
    - app.component.html
    - app.component.css
    - All other code
  - public-directory
    - Images
    - Possible other resources

Cli bundles all required libraries, no need to worry about system.js-configuration
- Huge relief

Main.ts just bootstraps, module is definened separately

Components by default use templateUrl styleUrls

Components

- Component is the very basic building block of Angular application
  - Extends html-vocabulary
  - Anything from very simple (HelloWorld) to the view of entire page
  - Appearance is described by
    - Template
    - TemplateUrl
- Behind each component there is an object (Class) that
  - Holds the data for the template
  - Holds the functionality for the template

Simple component with data binding

- Component has a selector: this will appear on the html-page
- Component has a template, or templateUrl that points to external html-fragment
  - Template may hold databindings
- Component is a decorator to a class holding the data to be presented by the template
  - Class may also contain methods for event handlers etc
- Top-level components need to be bootstrapped

simple.ts

```
import {Component} from '@angular/core';

@Component({
   selector: 'simple',
   template: '<input [(ngModel)]="data" /> {{data}}'
})
export class SimpleComponent {
   data: string = "Hello";
}
```

Module must import FormsModule from @angular/forms

Interpolation {{expression}}

- Interpolation may be used for one-way data-binding
  - Html-content
  - Property value
- The expression may be
  - Single property
  - Calculated value
  - Return value from a function call

Notice the ` -signs surrounding the multiline-template

interp.ts

```
@Component({
   selector: 'binding-demo',
   template: `
     <div><input value="{{data1}}" /></div>
     <div><input value="{{data2}}" /></div>

     <div>{{data1}}, {{data2}}</div>
     <div>{{data1+getSep()+data2}}</div>
`
})
export class BindingDemoComponent {
   data1: string = 'Hello1';
   data2: string = 'Hello2';

   getSep() {
      return '-';
   }
}
```

[One-way] data-binding from the source

- One-way data-binding may be done for
  - Property value
  - Attribute value

bind.ts

```
@Component({
   selector: 'binding-demo',
   template: `
     <div><input [value]="data1" /></div>
     <div><input [attr.value]="data2" /></div>

     <div>{{data1}}, {{data2}}</div>
`
})
export class BindingDemoComponent {
   data1: string = 'Hello1';
   data2: string = 'Hello2';

}
```

Note that the values in div are not changed when editing values in the input-fields

Two-way databinding

For the form fields the simplest way of doing two-way data-binding is with ngModel-directive as already seen:

simple.ts

```
import {Component} from '@angular/core';

@Component({
   selector: 'simple',
   template: '<input [(ngModel)]="data" /> {{data}}'
})
export class SimpleComponent {
   data: string = "Hello";
}
```

When ever the contents of the input-field change the data-member is also changed, when data-member is changed, the contents of input-field change accordingly

When using own components different notations may be used

- Let’s talk about this when we implement container component

Binding to (event)

- Event handlers may be implemented to the component class
  - click
  - blur
  - keyup
  - etc
- Any parameters may be passed to the event handlers
- $event passes the actual event-object

Bind.ts

```
@Component({
  selector: 'binding-demo',
  template: `
    <div><input value="{{data1}}" /></div>
    <div><input [value]="data2" /></div>

    <input type="button" (click)="change()" value="Change" />
    <input (blur)="changeTwo($event,'Hi',data1)“ />
`
})
export class BindingDemoComponent {
  data1: string = 'Hello1';
  data2: string = 'Hello2';

  change() {
    this.data1 = 'Hi1';
    this.data2 = 'Hi2';
  }

  changeTwo(ev, data, extra) {
    console.log(ev);
    this.data1 = data + "1";
    this.data2 = extra + “2";
  }
}
```

Component lifecycle methods

- Constructor can (and should) be used to initialize components own data
- How-ever child components are not necessarily initialized yet at that point
  - ngOnInit-should be used instead

- How-ever there are quite a few lifecycle hooks
  - ngOnInit - after the first ngOnChanges
  - ngOnDestroy - just before the component is destroyed.

  - ngDoCheck - developer's custom change detection
  - ngOnChanges - called when an input or output binding value changes

  - ngAfterContentInit - after component content initialized
  - ngAfterContentChecked - after every check of component content
  - ngAfterViewInit - after component's view(s) are initialized
  - ngAfterViewChecked - after every check of a component's view(s)

code
tutors

Container component

- Component may use other components
- Remember that the module must “declare” all components

Container.ts

```
@Component({
   selector: 'demo-contained',
   template: `<p>Hello world</p>
         `
})
export class ContainedDemoComponent { }

@Component({
   selector: 'demo-container',
   template: '<demo-contained>This is disgarded</demo-contained>'
})
export class ContainerDemoComponent { }
```

Passing data to component

- Input-decorator may be used to pass data from container to the child
  - Parameter specifies the attribute-name, if none is specified the property-name is used

Container.ts

```
@Component({
  selector: 'demo-contained',
  template: `<p>Hello {{greetingTarget}}</p>
        `
})
export class ContainedDemoComponent {
  @Input('target') greetingTarget: string="World";
}

@Component({
  selector: 'demo-container',
  template: `<demo-contained target='John'>This is disgarded</demo-contained>
        `
})
export class ContainerDemoComponent { }
```

Passing html-content

- ng-content -directive may be used as a placeholder for html content

Container.ts

```
@Component({
  selector: 'demo-contained',
  template: `<p>Hello {{greetingTarget}}, <ng-content></ng-content></p>
        `
})
export class ContainedDemoComponent {
  @Input('target') greetingTarget: string="World";
}

@Component({
  selector: 'demo-container',
  template: `<demo-contained target='John'>hows life</demo-contained>
        `})
export class ContainerDemoComponent { }
```

Component output, two-way databinding

- Event-based mechanism
  - Behind a property there can also be “change”-event
  - When using two-way databinding we actually subsrcibe and process the event automatically

Container.ts

```
import {Component,Input,Output,EventEmitter} from '@angular/core';

@Component({
   selector: 'demo-contained',
   template: `<p (click)="clicked()">count={{count}}</p>`
})
export class ContainedDemoComponent {
     @Input() count: number;
     @Output() countChange: EventEmitter<number> = new EventEmitter();

     clicked() {
          this.count++;
          this.countChange.emit(this.count);
     }
}

@Component({
   selector: 'demo-container',
   template: `<demo-contained [(count)]='myCount'>Ignored</demo-contained>
          <p>myCount is {{myCount}}`
})
export class ContainerDemoComponent {
   myCount = 3;  //Automatically updated
}
```

Event-handler for child component events

- Event-emitter member is basically an event that we may process
- The value emitted is passed as the parameter to the event handler

Demo-contained is the same as on the previous slide

```
@Component({
   selector: 'demo-container',
   template: `<demo-contained [count]='myCount' (countChange)="myHandler($event)">
         Ignored</demo-contained>
          <p>myCount is {{myCount}}`
})
export class ContainerDemoComponent {
   myCount = 3;

   myHandler(ev) {
      this.myCount = ev;
   }

}
```

Structural directives and templates (ngIf)

- Structural directives are most often used in their “asterisk-form”
  - *ngIf, *ngSwitch, *ngFor
- Angular on the background compiles the asterisk form into template form
  - Template element in in html has the display-style property set to none, so it is invisible and doesn’t take up space
  - Data-binding still works

Structural.ts

```
import {Component} from '@angular/core';

@Component({
   selector: 'structural',
   template: `
        <p *ngIf="doShow">Hello world</p>
        <ng-template [ngIf]="doShow">
            <p>Hello world</p>
        </ng-template>
        `
})
export class StructuralDemoComponent {
   doShow: boolean = true;
}
```

The paragraph only appears in dom-model when doShow is true.

Both versions operate exactly the same

ngSwitch

- ngSwitch is bound to property to be watched
- ngSwitchWhen is kind of an “if” for specific value
  - Again template translation occurs

Structural.ts

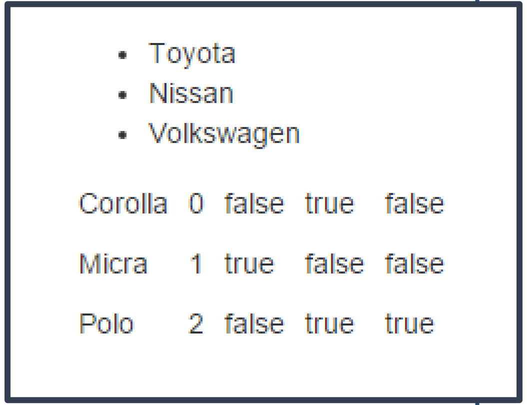
```
@Component({
   selector: 'structural',
   template: `
         <p (click)="count=count+1;">Count = {{count}}</p>
         <div [ngSwitch]="count">
           <p *ngSwitchCase="0">Starting, click the count</p>
           <p *ngSwitchCase="1">Again...</p>
           <p *ngSwitchCase="2">And again...</p>
           <p *ngSwitchDefault>All done</p>
         </div>
         `

})
export class StructuralDemoComponent {
   count: number = 0;
}
```

ngFor

- Repeat the element for each member within a collection
- Local variables may be used
  - let variable_name= when declared (earlier #variable_name=)
  - Index, odd, even, last

Structural.ts

```
@Component({
   selector: 'structural',
   template: `<ul><li *ngFor="let car of cars">{{car.make}}</li></ul>
         <table>
         <tr *ngFor="let car of cars;let i=index; let o=odd; let e=even;let l=last">
             <td>{{car.model}}</td>
                 <td>{{i}}</td><td>{{o}}</td>
                 <td>{{e}}</td><td>{{l}}</td>
         </tr></table>`
})
export class StructuralDemoComponent {
   cars = [
     { make: 'Toyota', model: 'Corolla' },
     { make: 'Nissan', model: 'Micra' },
     { make: 'Volkswagen', model: 'Polo'}
   ];
}
```



```
• Toyota
• Nissan
• Volkswagen

Corolla  0  false  true   false
Micra    1  true   false  false
Polo     2  false  true   true
```

Angular 17, Built in control flow

- Simplify use of structural directives
- @if
- @switch
- @for

```
export class AppComponent {
  public option="uninitialize";
  public fruits=[{id:1,name:"Bananas"},{id:2,name:"Apples"},{id:3,name:'Oranges'}];
}
```

```
@if(option=="enter"){<p>Hello</p>}
@else if(option=="leave"){<p>Bye</p>}
@else{<p>Not initialized</p>}

@switch(option){
  @case("enter") {<p>Hello</p>}
  @case("leave") {<p>Bye</p>}
  @default{<p>Not initialized</p>}
}
<input type="button" value="Enter" (click)="option='enter'" />
<input type="button" value="Leave" (click)="option='leave'" />
<input type="button" value="Reset" (click)="option=''" />

@for(fruit of fruits; track fruit.id){
  <p><span>{{fruit.id}},{{fruit.name}}</span></p>
}
```

Pipes

- Pipes modify data before it is displayed
- Declared in angular/common -module
  - AsyncPipe (async)
  - CurrencyPipe (currency)
  - DecimalPipe (number)
  - DatePipe (date)
  - JsonPipe (json)
  - LowerCasePipe (lowercase)
  - PercentPipe (percent)
  - SlicePipe (slice)
  - UpperCasePipe (uppercase)
- Pipe-specific extra parameters may be passed for processing
  - Colon (:) after the pipe name

Dependency injection in Angular

- Dependency injection makes it easy to access globally available objects
- In the most typical form the injectable objects are just added to the constructor of the Component
    - But there are quite a few variations
    - And different results may occur depending on the injector hierarchy
    - Somewhere in the injector-hierarchy a provider for the injectable object must be defined
- In fact using the injectable objects should require a little thought
    - As should implementation of injectables

Injectable value

- Constants that are used throughout the application may be provided by the application bootstrap

DemoApp.ts (module descriptor)

```
providers:[
    {provide:‘Greeting',useValue: ’Hello world'},
]
```

- Any component may inject the value to its constructor
  - Inject the injector and get the value from injector
  - OR just @Inject the “Greeting” value

```
constructor(injector:Injector, @Inject("Greeting") greet) {
     let s = injector.get("Greeting");
     console.log(s,greet);
}
```

What can be injected/provided

- Values, functions, objects

DemoApp.ts

```
providers: [
    {provide:"Greeting", useValue: 'Hi there' },
    {provide:"DoIt", useValue: function (x) { console.log(x); } },
    {provide("GObj", useValue: {x:5,y:10}}
]
```

- And the components constructor

```
constructor(@Inject("Greeting") greet,@Inject("DoIt") doit,@Inject("GObj") obj) {
    console.log(obj.x,obj.y, greet);
    doit("Injected function");
  }
```

Impelenting injectable classes

- @Injectable-decorators declares an injectable class

Injects.ts

```
import {Component,Injector,Inject, Injectable} from '@angular/core';

@Injectable()
export class SomeStuff{
   data: string = "Hello";

   showData() {
      console.log(this.data);
   }
}

@Component({
   selector: 'demo-inject',
   template: '<p>{{someStuff.data}}</p>',
   providers:[SomeStuff]
})
export class InjectDemoComponent {
   constructor(public someStuff: SomeStuff) {
      someStuff.showData();
   }

}
```

Provider must be available in the injector-hierarcy of the component

Deferrable views (lazy loading)

- Introduced at v17, stable at v18
- Template may hold @defer{<OtherComponent />} –block
  - Separate chunk will be generated for the code of other component
  - And the code will be loaded only when needed
- So when the code is loaded
  - If nothing is specified for @defer, the code is loaded when browser becomes idle
  - You can specify conditions and triggers for @defer

DeferredComponent is loaded when checkbox is checked

```
<input type="checkbox" [(ngModel)]="loadNow" />

@defer(when loadNow){ <deferred /> }
```

Demo/Excercise

- ng new DemoApp
- cd DemoApp
- ng generate component deferred
- ng serve –open

- If you are not familiar with new notations of @if, @for and @switch experiment a little with them
- Use @defer to load the DeferredComponent
  - Check the console output to see that a separate block is generated
  - Create a condition or trigger for @defer to take control of when the loading happens
    - https://angular.dev/guide/defer
  - Check the network monitor to see that the loading actually happens at separate step

Routing

Routing overview

- In Angular-terminology routing means navigating between different views of our application
- The “main-view” is just a layout-template containing items common to entire application
  - Title-bar
  - Main navigation
  - Footer
- The main-view also contains a placeholder for the actual views
  - <router-outlet>
- The router is then configured so that it knows which component to display in the placeholder
  - Based on the url-pattern
- Routing has changed a lot between different versions of Angular
  - Different examples might still be around

Routing configuration

- Routing is implemented into a separate module
  - Which must be imported to app-module
  - Below is a simple implementation of routing module, angular cli creates a “full” module constructs
- Main component is typically just a layout-template
  - Contains common items for all pages and
  - `<router-outlet />`
- Routes are configured to a separate Angular module
  - That must appear in the imports of the main module

```
import {Router, Routes, RouterModule} from '@angular/router';
import {ModuleWithProviders} from '@angular/core';

const routes=[
    { path: 'cars/:id', component: CarDetail },
    { path: ‘cars', component: CarList},
    { path: ‘home', component: HomeView},
    { path: ‘’, redirectTo: ‘home’, pathMatch: ‘full'},
    { path: ‘**', component: NotFound },
];

export const MyRoutingModule:ModuleWithProviders
       =RouterModule.forRoot(routes,{useHash:true});
```

:id marks a route parameter, you can append ? if the parameter is optional

Do not pass the second parameter if you want to use html5-style navigation

Child routing / nested routing

- Basically any component may hold “router-outlet”
  - There should also be a matching child route configuration for that component

```
@Component({
   selector: 'demo-child',
   template: `<h2>Manufacturers</h2>
              <router-outlet></router-outlet>
          `,
})
export class ManufacturersComponent { }

const routes=[
    { path: 'cars/:id', component: CarDetail },
    { path: ‘cars', component: CarList},
    {
      path: ‘manufacturers',
      component: ManufacturersComponent,
      children: [
         { path: '', component: ManuList },
         { path: ‘:id', component: ManuDetail }]
    },
    { path: ‘home', component: HomeView},
    { path: ‘’, redirectTo: ‘home’, pathMatch: ‘full'},
    { path: ‘**', component: NotFound },
];
```

Special case, componentless route

```
{ path: 'parent/:id', children: [
    { path: '', component: FirstChild },
    { path: '', component: SecondChild, outlet: 'second' } ]
}
```

- /parent/:id doesn’t display any component
  - Or it displays both FirstChild and SecondChild
- Now there must be two router-outlets at template

```
   <main>
     <router-outlet (activate)="routerActivate($event)" />

     <router-outlet name="second" />
   </main>
```

Custom routes

- You can define complex rules to match URL-pattern to a component
  - Use regular expressions
  - Use different conditions
- Instead of path add matcher-property to Route-element
  - Function that takes url-segments as parameters
  - Return null if this rule doesn't apply
  - Return UrlMatchResult-object if the rule applies
    - At least with consumed property

```
{path:"books/:id",component:DetailComponent},

{
  matcher: (segments) => {
    if (segments.length<1) return null;
    if (segments[0].path!="error") return null;
    return{
      consumed:segments,
      posParams:{cause:segments[1]}
    }
  },
  component: ErrorComponent
},
```

You wouldn't actually need custom matcher for this. path:'/error/:cause' would do the same

Navigating to routes

- Traditional a hrefs work of course provided we know the location strategy

```
<a href="#/cars/4">Goto car with id 4</a>
```

- routerLink creates an href for given route name

```
<a routerLink="/cars">List of cars</a>
<a [routerLink]="['/cars',4]">Car 4</a>
```

- In the code you may call router:Router navigate-method

Component code

```
constructor(private router: Router){}

navToList(){
    this.router.navigate(['/cars']);  // Preferred
    /* You can also navigate by url
    this.router.navigateByUrl("/cars");
    */
}

navToSingle(id){
    this.router.navigate(['/cars',id]);
}
```

The target may implement:

```
constructor(private router:Router,
        private route:ActivatedRoute){
}

ngOnInit(){
   this.route.paramMap.subscribe(
      params =>  this.carID=params.get('id')
   );
}
```

Catching route params

- Previous slide showed how to subscribe the paramMap for Activated route
- If you don’t intend to change the route parameter while the component is displayed, you can also use
  - this.carId=this.route.snapshot.paramMap.get(”id”)
- Or you can add providers to your router module

```
  providers: [
    provideRouter(routes, withComponentInputBinding()),
  ]
```

- And use @Input properties at target component

```
    @Input()
    set id(cid: string) {
      this.carIid = cid;
    }
```

Extra notes about navigation

- When navigating you perhaps most often use absolute paths
  - Begin the url with slash /
- Occasionally (especially with child routing) you may wish to use relative paths instead
  - No leading slash
  - Or ./xxx
    - Can also be for example ../yyy
  - When using router.navigate add second parameter
    - this.router.navigate([’xxx’], {relativeTo:this.route});
- Optional parameters must be passed in an object in link parameters array
  - this.router.navigate([’/zzz’, {some:’optional parameter’}]);

Router events

- The router can signal various events
  - https://angular.dev/guide/routing/router-reference
  - These can be subscribed in the code

```
ngOnInit(){
  this.router.events.subscribe(e => console.log(”Router event",e));
}
```

Remember to unsubscibe the returned subscription if this is not app-component

- Often it is enough to know which component has been activated

```
<router-outlet (activate)="routerActivate($event)" />
```

```
  routerActivate(event:any){
    console.log("Router activate",event);
  }
```

Event is actually reference to the activated component

Router guards

- Guards control the routers behavior, most often “Can we continue navigating to this route?”
- Guards are described as interfaces and implemented as injectables
  - CanActivate-interface -> canActivate-method
  - CanDeactivate-interface -> canDeactivate-method

  - CanActivateChild-interface -> canActivateChild-method
  - Resolve-interface -> resolve-method (load data before route is activated)
  - CanLoad-interface -> canLoad-method (can a module be loaded asynchronously)
- Or (currently by default), they can be implemented as functional guards
  - Just implement a single function that takes the required parameters

After implementing the guard it must be applied to the route-configuration:

```
{path:’test’,component:TestComponent,
          canActivate: [… Array of guard-types …],
          canActivateChild: [… Array of guard-types …]
}
```

Exercise – Add routing

```
ng generate module app-routing --flat --module=app
```

- Study the code generated + app-module.ts
- Add routing configuration
  - BasicsMainComponent should be default component
  - Url ’calc’ should lead to CalculatorComponent
  - `imports: [RouterModule.forRoot(routes)],`
  - `exports: [RouterModule],`
- Add navigation to nav-section of app-component.html
- Add router-outlet to main-section of app-component.html

Exercise – route parameters

- Extend router configuration
  - ’books’ => BookListComponent
  - ’books/:id’ => BookDetailComponent
- Navigate to BookDetail from BookList
  - Add a routerLinks to id-columns of booklist
  - Add click-handlers to title columns of booklist
- Catch id-paramater at BookDetail, try these both approaches
  - Subscribe paramMap at ngOnInit
  - Add @Input property
- Add another @Input property called ”some”
  - Try passing query parameter ?some=value to book detail
- Try adding optional route parameter to the router configuration

Exercise – Router events

- Subscribe the router events at Calculator ngOnInit
  - Just console log the event
- Navigate to the calculator, then away from calculator
  - What do you notice?
- Add ngOnDestroy to calculator
  - Unsubscribe the events

- Add activate event hander to router-outlet
  - Console.log for the event parameter is again enough

Exercise - guards

```
ng generate guard basics/ActivateGuard
```

- Console log both parameters at guard
- Assign canActivate-guard to Calculator-route
- Experiment with true/false return values
- Also

```
const router=inject(Router);
return router.parseUrl("/books")
```

- Create Deactivate guard in the same manner
- Also ng generate service basics/DirtyService
  - DirtyService should hold public property isDirty=true
- Display isDirty at BasicsMain and add a button that toggles isDirty property
- Inject DirtyService to to Deactivate guard and prevent deactivation based on the value of isDirty
- Add canDeactivate guard to BasicsMain -route

Modules

Designing and implementing

Lazy loading

Implementing libraries

Designing modules

- Small Angular application may just use single module that declares all components, pipes, services and directives
  - And perhaps a router module with app-module
- Bigger the application grows more important it becomes to implement separate feature modules
  - Module for working with each entity type is a good starting point
- If you are uncertain whether you need modules you should at least group components to separate subdirectories
  - Again a subdirectory for each entity type
  - Perhaps a separate subdirectory for ”utilities”
    - Custom pipes and directives
    - Helper -functions

Implementing modules

```
ng generate module MyModule --module=app --routing
```

Optional: which module imports the new module

Optional: will the module implement child routing

- Needed modules must be imported by other modules
  - Often app-module imports all other modules
- Each module must also import all necessary Angular modules

```
@NgModule({
  declarations: [],
  imports: [
    CommonModule,
    FormsModule
  ]
})
export class MyModuleModule { }
```

Perhaps the name given at ng generate should only be ”My”

Child-routing

- Modules can also implement child-routing

```
@Component({
   selector: 'demo-child',
   template: `<h2>Demonstrations</h2>
              <router-outlet></router-outlet>
          `,
})
export class ChildDemoComponent { }

const childRoutes: Routes = [{
   path: ‘manuroute',
   component: ChildDemoComponent,
   children: [
     { path: '', component: ManuList },
     { path: ‘:id', component: ManuDetail }]
}];

@NgModule({
  imports: [
   RouterModule.forChild(childRoutes)
  ],
  exports: [
   RouterModule
  ]
})
export class ChildRoutesModule { }
```

Notice!

Lazy loading

- Modules that are imported by app-module are loaded immediately when the application loads
  - In big applications this may cause a long delay before the application launches
- The modules that are not required immediately may be ”lazy loaded” in the background
  - Routing configuration

```
{path: ’some’,
      loadChildren: () => import(’./some/some.module’)
        .then(m => m.SomeModule)
}
```

Exercise – books to module

- ng generate module books
- Copy BookList, BookDetail and BookService into books-directory
- Declare BookList and BookDetail at BooksModule
  - Remove the declaration from AppModule
- Import BooksModule to AppModule

- And everything should still work….

Exercise – Lazy loading module

- ng generate module users --route users --module app
- ng generate component users/UserList --module users
- ng generate component users/UserDetail --module users
- Add children to user-routing modules
    - ‘’ => UserList
    - ‘:id’ => UserDetail
- And <router-outlet /> to UsersComponent

- Use developer tools networking to check when the users module loads

Excercise – some work with modules

- ng generate service users/User

```
import { Injectable } from '@angular/core';
import {HttpClient} from '@angular/common/http'

@Injectable({
  providedIn: 'root'
})
export class UserService {

  constructor(private http:HttpClient) { }

  public getAll(){
    return this.http.get("/api/persons");
  }

  public get(id:any){
    return this.http.get("/api/persons/"+id);
  }
}
```

- Inject UserService to UserList and console log the result from getAll at ngOnInit
  - What do you notice
- Provide HttpClient from UserModule

  ```
  providers: [ provideHttpClient() ],
  ```

- Remove the parameter object from @Injectable and add provider for UserService to UserModule
  - Now it should work….
- Display users in the user list and navigate to user detail from the list

Excercise – Some work with modules

- Inject ActivatedRoute and UserService to UserDetail
- Subscribe the route paramMap at ngOnInit
  - Then get userid from paramMap
  - And call userService.get(userId)
  - Save the user to a public property
- Add input field with databinding to user.username
  - What do you notice
- Add FormsModule to the imports of UserModule

- Extra:
  - Do you need UserComponent?
  - Could you use componentless routing instead?

Implementing libraries

- Library is a collection of angular objects that can be imported to a project
  - Like npm modules
  - They can actually be published to npm repository
- We need to create Angular workspace which may contain several projects
  - Applications
  - Libraries

Exercise – new Library

- ng new lib-workspace --no-create-application
- cd lib-workspace
- ng generate library pipe-lib
- ng generate pipe plural

```
@Pipe({
  name: 'plural'
})
export class PluralPipe implements PipeTransform {

  transform(value: number, terms:[string,string]): string {
    if (value==1) return value+' '+terms[0];
    return value+' '+terms[1];
  }

}
```

Exercise – Testing library

- Since our original application is not in the same workspace we need to use a nasty import in book-app/app-module
  - `import {PipeLibModule} from '../../../lib-workspace/dist/pipe-lib’;`
- But after that we can use plural pipe in basics-main

```
<p>Plural {{1 | plural:['song','songs']}}</p>
```

- Go back to lib-workspace
- ng generate app TestApp
- You can just
  - `import {PipeLibModule} from 'pipe-lib’;`
- Use the pipe in app.component.html
- And ng serve TestApp --open

Rx.js

Rx.js and observables

- Angular uses Rx.js (Reactive Extensions) for asynchronous tasks and some data manipulation
  - EventEmitter
  - Forms-module
  - Http-module
- Basically Rx.js translates the Promise pattern of asynchronous methods into Observable pattern
  - Where instead of getting one value from async call you are prepared to receive a stream of data
  - And data may be received by several interested parties
- The data from the stream may be ”piped”
  - Manipulated somehow as it arrives
- So we have an observable object
  - We subscribe the information from that object
  - And unsubsribe when we no longer need the data

Creating observables

- Observables may be created from different sources
  - From promise - rxjs.fromPromise(promiseObject)
  - Interval – rxjs.interval(time-out-in-ms)
  - Dom event – rxjs.fromEvent(domElement,event-name)
  - Ajax request – rxjs.ajax(url)
  - From stream of data – Observable.of(list-of-values)
- Most importantly observables provide the subscibe-method
  - Callback provided processes the value received from observable
  - Separate callbacks may be provided (currently deprecated)
    - Error handler
    - Completed handler (Observable will provide no more data)
  - Or pass a subscriber object with methods:
    - next(data)
    - error(err)
    - complete()

Manipulating data

- Rx.js comes with dozens of operators that are used to manipulate data
  - In rxjs/operators –module
- Quite a lot to learn…
- Combination (10+ methods)
  - Combine data from different observables
- Conditional : defaultEmpty, every
- Filtering (some 20 methods)
  - Filter, first, last, skip, take….
- Transformation (some 20 methods)
  - Map, reduce, buffer…

(Somewhat) typical use

- Here we get an array of data
- Pipe the date through operators
  - concatAll – process each item individually
  - Filter – operate on items matching the criteria
  - Map – produce a new item that is used later on

```
http.get<Greeting[]>("/api/demo")
    .pipe(
        concatAll(),
        filter(g => g.target.includes("o")),
        map(g => g.greeting)
    )
    .subscribe({
        next(s){ this.greetings.push(s); },
        error(err){ console.log("ERROR",err); },
        complete(){ console.log("No more data"); }
    });
```

Exercise - operators

- Add testRxjs-method to UserService
  - It doesn’t need to return anything
  - Make http.get to query all persons
    - You will need to create Person-class if you haven’t already done so
  - And use the operators as demonstrated on previous slide
- Add a button to UserList with which you can launch the test

- Modify UserService getAll so that it stores the users received to a property in UserService
  - Modify testRxjs so that it creates Observable from an array instead of making http.get -call

State Management

Component state vs. Application State

State Management Libraries

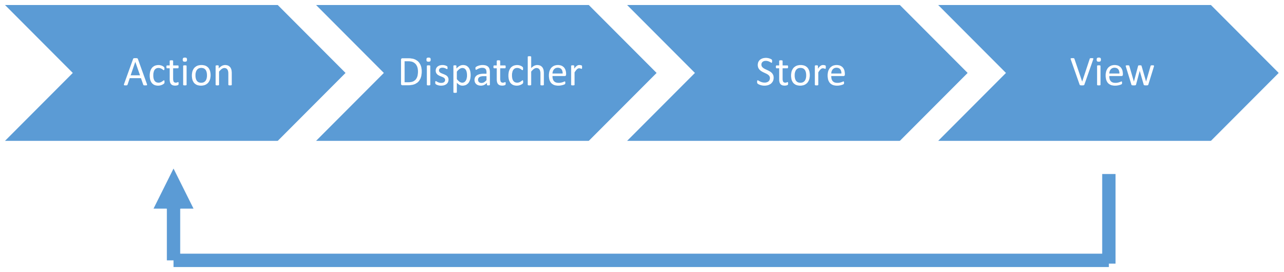
Redux and NgRx

Component State vs Application state

- Component state is reflected by the properties the component class holds
- Application state consists of data items several components need
  - Application state can be managed with injectable Services to certain extent
- In very complex situations managing application state with just services becomes too difficult (SHARI)
  - Shared: state that is accessed by many components and services
  - Hydrated: state that is persisted and rehydrated from external storage.
  - Available: state that needs to be available when re-entering routes.
  - Retrieved: state that must be retrieved with a side-effect.
  - Impacted: state that is impacted by actions from other sources.

FLUX Architecture

- FLUX-architecture describes application state management
  - Not a framework, just a pattern
  - Can be implemented with convenient techniques
- Unidirectional dataflow
  - Store(s) hold the data and know how to process different actions against it
    - No getters and setters, just callbacks through which the dispatcher passes the action to the store
  - When the store manipulates data it creates an event which can then be processed by the view(s)
  - The view then may generate new action(s)



What is Redux

- Redux is a state-container for JavaScript applications
  - Can rather well be used as a Store in FLUX-architecture
  - Simplifies passing data between components
- Lightweight and simple to use
  - redux npm-module
  - Actual data that is held by the container may be anything
  - We just implement a function that takes state and action as parameter
    - And returns the new state based on the action
  - Elsewhere we may subscribe to the changes in the store
  - And dispatch actions to the store
- NgRx is built on Redux
  - Same basic concepts: Store, Reducers, Actions
  - But with some Angular specific extensions

Getting started with NgRx

- ng add @ngrx/store@latest --no-minimal
  - Installs necessary npm packages (+ modifies package.json)
  - Creates reducers-directory and index.ts there in
  - Modifies app-module with import to StoreModule.forRoot
- We are going to create an application for managing authors
  - ng generate component authors/Authors
  - ng generate component authors/AuthorList
  - ng generate component authors/AuthorDetail
  - Modify AuthorsComponent to display AuthorList and AuthorDetail side-by-side
  - Add top menu navigation to AuthorsComponent

Store (state)

- Our store will contain array of authors and selected Author
- Create entities\author.ts
  - export interface Author

    id:number, firstName:string,lastName:string
- Modify reducers/index.ts
- Check AppModule
  - `StoreModule.forRoot({authors:authorsReducer},{}),`

```
export const authorsState:ReadonlyArray<Author>=[];
export const setAuthors=createAction("[Set Authors",props<{authors:Author[]}>());
export const authorsSelector = createFeatureSelector<Author[]>('authors');


export const authorsReducer = createReducer(
  authorsState,
  on(setAuthors, (state, {authors}) => authors),
);
```

This is index.ts

Dispatch and subscribe

- Now AppComponent (for example) can read authors from server and dispatch setAuthors –action

```
  ngOnInit(){
    this.http.get<Author[]>("/api/authors").subscribe(authors => {
      console.log("Authors",authors);
      this.store.dispatch(setAuthors({authors}))
    });
  }
```

- Elsewhere (AuthorsList) the data may be consumed

```
  public authors:Observable<Author[]>;

  constructor(private store:Store<{authors:ReadonlyArray<Author>}>){
    this.authors=this.store.select(authorsSelector);
  }
```

- And displayed

```
<h2>Authors</h2>
<p *ngFor="let a of authors | async">{{a.lastName}}</p>
```

Exercise - selectedAuthor

- Create similar constructs to index.ts for selectedAuthor as on previous walkthrough
  - Initial state holding Author
  - setSelectedAuthor –action
  - selectedAuthorSelector –selector
  - selectedAuthorReducer
- Add reducer to StoreModule.forRoot
- Create click-handler to authors name at AuthorList
  - Dispatch setSelectedAuthor –action
- Modify AuthorDetail to display selectedAuthor

Exercise – create Author

- Add createAuthor-action to authorsReducer
- Add Create-button to AuthorDetail
  - Click handler uses http.post to create new Author
  - When http.post completes dispatch createAuthor-action

- Extra
  - Also implement Save and Delete

Internationalization

… and localization

Formatting data

Translations

Overview

- Localization means that you build your project with specific language settings
  - Through internationalization (I18N) you prepare your project to be distributed to several countries around the world
  - You localize your application to several languages
- It means translating all the strings to to different languages and also formatting the data in locale specific way
- Angular provides good tools for I18N but the process still requires heaps of work
- And also some planning
  - Are we sending emails, how are those translated
  - Long strings perhaps with inserted variables
  - ”Article” type of pages, Help-pages
  - Convert prices to different currencies
  - Do we need time zones
  - Who will do the translations
  - Are we using images that have culture specific meaning
  - How about languages that are read from right to left

Getting started with Localization

- ng add @angular/localize
  - Modifies package.json, tsconfig.json and main.ts
- Choose the default locale
  - Set the LOCALE_ID token that the pipes that format data use
- Possibly register the required locales
  - Other than en-US

main.ts

```
/// <reference types="@angular/localize" />

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

import { LOCALE_ID } from '@angular/core';

import locFi from '@angular/common/locales/fi';
import {registerLocaleData} from '@angular/common';
registerLocaleData(locFi);


platformBrowserDynamic().bootstrapModule(AppModule,{
  providers: [{provide: LOCALE_ID, useValue: 'fi-FI' }]
}).catch(err => console.error(err));
```

Register locales, depending on angular.json configuration these may or may not be needed

Angular.json

- All supported locales must be specified at angular.json
- Now we can add i18n-tags to templates and run

  ng extract-i18n --output-path src/locale --out-file messages.fi.xlf

Within ”projects”:{”BookApp”:{….}}

```
      "i18n": {
        "sourceLocale": "en",
        "locales": {
          "fi": {
            "translation": "src/locale/messages.fi.xlf",
            "baseHref": ""
          }
        }
      },
```

Default doesn’t have to be registered

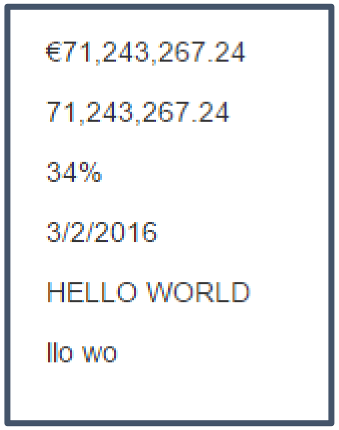
Translations file name

code tutors

Formatting data

- By default you should use pipes for formatting data
- All pipes use LOCALE_ID for default
- But all relevant pipes can also be forced to format according to specific locale

```
@Component({
   selector: 'pipe-demo',
   template: ` <p>{{num | currency:'EUR’:’symbol’:'1.2-2’:’en’}}</p>
                <p>{{num | number:'1.2-2’:’en’}}</p>
                <p>{{num2 | percent:'1.0-0’:’en’}}</p>
                <p>{{dt | date:'d.M.y’:’en’}}</p>
                <p>{{str | uppercase}}</p>
                <p>{{str | slice:2:8}}</p>
          `
})
export class PipeDemoComponent {
   num: number = 71243267.235433;
   num2: number = 0.336234;
   str: string = 'Hello world';
   dt: Date = new Date();
}
```



```
€71,243,267.24

71,243,267.24

34%

3/2/2016

HELLO WORLD

llo wo
```

Translations

- After enabling localization we can add i18n-tags to templates to mark items that need to be translated
  - <p i18n>Hello</p>
  - <img title=”Logo” i18n-title src=”logo.png” />
  - There are quite a few options on how i18n-tags can be used, study them thoroughly before starting the translation work
- You can also mark strings in your code to be added to the translation files
  - title=$localize`String to be translated`
  - $localize can only be used with template literals (`xx`)
- Metadata is important, it should provide information to the translator on where and how the string is used
  - i18n=”Some description”

Distributing

- First you need to add to angular.json
  - architect:{build:{options:{   ”localize”:true   }}}
- Then ng build
  - Separate directory is created for each locale
- Now it is time for server configuration….
  - Follow instructors lead

Progressive Web Applications

What is PWA

Service Workers

What are PWA and Service Workers

- PWA allows application to be ”installed” to the browser so that it can also be used offline
  - Consider it to be cached + some extra functionality
- PWA relies on Service Workers
  - Client side proxy before http-requests
  - When network is not available Service Worker provides cached data
- When the application just displays data getting started with PWA and Service Workers is rather straight-forward
  - If we intend to modify data offline the architecture needs to be designed
  - Offline data store is also needed
    - Indexed db provided by browser?

Walkthrough – getting started

- ng add @angular/pwa
- ng build
- Run the application through localhost:9000
  - Have you configured the server already?
- After browsing through some of the pages of the application
  - Use developer tools/networking to disable network
  - Reload the application
    - How do the network requests appear
  - Also study developer tools/application/service workers

Next steps

- Getting started is easy
  - Next steps are pain
- By default you always work with ”installed” version of the application
  - You should implement a feature with which you can check if there is a new version available on server
    - And possibly force ”refresh”
- Configure service workers
  - What resources should be handled by service worker
  - ngsw-config.json
- Should application behave differently when offline?
  - Disable some features
  - Or operate with offline data
- Back to first step
  - Do you really need Service Workers?
  - Complicated issues to tackle
  - How to make sure user always has latest valid
    - Application
    - Data

Walk-through - Indexed db

- One valid reason to use PWA –techniques is for applications that often need to operate offline
  - Collect data in desert or wildernes where network is not available
- Then you need to store the data to the browser
  - Until network becomes available
  - And sync the data afterwards
- Follow the instructors lead to populate some data to indexed db

Server side rendering

Why render on server

Angular Universal

Server side rendering

- When using server side rendering the server creates static html content from the application
  - The static content is loaded to the browser
  - When the static content is displayed the application launches and uses content from the web page
    - Hydration
- Server side rendering can be used for
  - SEO, search engines can crawl through static site
  - Improve performance
    - Especially the first page load
- Limitations, not everything can be done by the server
  - The application shouldn’t use browser resources (window, location etc)
  - Applications using Service Workers bypass server rendering after initial render
  - Also applications that use i18n cause some issues
- Saddly our BookApp uses quite a few of those limiting features….
  - Server side rendering is best suited for marketing oriented sites to improve performance and SEO

Exercise - walkthrough

- ng new SsrApp
  - Include routing
- cd SsrApp
- Ng generate component FirstPage
- Ng generate component SecondPage
- Configure routing to FirstPage and SecondPage
- Modify AppComponent html to display links to FirstPage and SecondPage
  - And <router-outlet />
- Add client hydration to module
  - providers[provideClientHydration()]
- npm run dev:ssr
  - Test the application,
  - Again see developer tools/Networking, try slower connection
  - View the page source

Prerender

- ng run SsrApp:prerender --routes / /second
- Check the dist folder
- Again, follow instructors lead for server configuration
    - Try through localhost:9000

Security of SPA-application

Security

- You want to protect your data
  - Especially secure the RESTful interface
  - But also
    - Credentials in application logic
    - Secure the server (technical credentials)
- You want to protect the data communication
  - HTTPS
- You might even want to protect the application
  - Logic
  - Templates
  - Resources
- Security measures you choose must come from the requirements of your solution
  - Traceabilty
  - Undeniability
  - Usability

Protecting the RESTful services

- You have to select the authentication method depending on
  - The possibilities your sever environment gives you
  - Your application and its requirements
- In your code
  - You may pass the Authorization-header with your request
  - You may pass authorization information in a cookie generated in user login
  - You may pass authorization information in the body of each request

- NEVER store technical credentials for authentication into the JavaScript-code loaded to the browser

Protecting the application itself

- If you don't want the unauthenticated users to access the
  - Templates
  - JavaScript
  - On other resources
- You have to
  - Design the distribution directory structure so that unauthenticated users can only access the "Login application"
    - Login application then gives access to the rest of the application
  - Figure out how to pass authorization token from login application to the actual application
- This requires some server specific configurations

Angular features for security

- Router-guards
  - Redirect to login page until successful login
- CanActivate
  - Prohibit view activation until successful login
- Extend RouterOutlet to your own view placeholder
  - That displays the login page until successful login
- http-request options
  - Pass Authorization header

Exercise

- Implement Login component
  - You may have input fields for credentials but they are not necessary
  - You just need a button whose click stores “login information” to localstorage
  - Experiment with different possibilities to redirect to login form until successful login
- Verify that your server requires basic authentication for the RESTful interface
  - Pass the authentication information to the server with HTTP-requests