

Angular Module

1 ALLGEMEIN	2
1.1 Grundprojekt	2
1.1.1 app.component.html	2
1.1.2 app.module.ts	2
1.1.3 app.routing.module.ts	3
2 MODELS	4
3 CORE	5
3.1 Merkmale	5
3.2 Erzeugen	5
3.3 Verwenden	5
4 SHARED	7
4.1 Merkmale	7
4.2 Erzeugen	7
4.2.1 shared.module.ts	7
4.3 Verwendung	7
5 FEATURE MODULE EAGER	8
5.1 Erzeugen	8
5.2 Routing	8
5.2.1 feature-eager-routing.module.ts	8
5.2.2 feature-eager.module.ts	8
5.2.3 app.module.ts	9
5.3 Verwendung von Core	9
5.4 Verwendung von Shared	9
6 FEATURE MODULE LAZY	11
6.1 Erzeugung	11
6.2 Routing	12
6.2.1 feature-lazy-routing.module.ts	12
6.2.2 feature-lazy.module.ts	12
6.2.3 app.module.ts	12
6.2.4 app-routing.module	12
6.3 Verwendung von Core	13
6.4 Verwendung von Shared	13
6.5 Eigenes Service	13
7 VERGLEICH EAGER/LAZY	15
8 ZUSAMMENFASSUNG	16
8.1 Dateistruktur	16
8.2 Moduleigenschaften	16
9 PRELOAD STRATEGIES	17
9.1 Preload all	17
9.2 custom strategy	17

1 Allgemein

Vor allem bei größeren Projekten ist es wichtig, die vielen Komponenten, Services,... vernünftig zu strukturieren. Dazu werden Module verwendet.

Dabei wird folgende Grundstruktur vorgeschlagen:

models	Hierher kommen die Model-Klassen
core	Hierher kommen Services, Interceptors und Guards
shared	Hierher kommen gemeinsame Komponenten und Pipes
feature	Feature Modules sind Module, die Teile einer App kapseln, z.B. einzelne Bereiche, die aus mehreren Seiten bestehen (OrderList mit Order Detail Ansicht und Order Creation Page). Diese können eager oder lazy geladen werden.

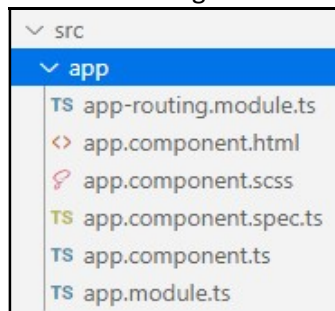
1.1 Grundprojekt

Neues Projekt erzeugen, incl. Routing.

```
C:\_PR\Angular>ng new Angular8AppRouting
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use?
CSS
> SCSS [ https://sass-lang.com/documentation/syntax#scss ]
Sass [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
Less [ http://lesscss.org ]
Stylus [ http://stylus-lang.com ]
```

Im folgenden Beispiel sollen nur verschiedene Module, Komponenten,... ohne Funktionalität erzeugt werden, um sich auf den Modulaufbau konzentrieren zu können.

Es entsteht die gewohnte Struktur, jedoch aufgrund des Routings inkl. der Datei **app-routing.module.ts**



Routing wird noch einmal separat besprochen, für das Verständnis der Module sollten aber die unten stehenden Erklärungen reichen.

1.1.1 app.component.html

Die Datei app.component.html wieder entrümpeln, nicht jedoch **<router-outlet>**.

```
<h1>Angular Modules</h1>

<router-outlet></router-outlet>
```

1.1.2 app.module.ts

Sieht aus wie gewöhnlich, jedoch mit **AppRoutingModule**.

Zur Sicherheit auch FormsModule und HttpClientModule importieren.

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

1.1.3 app.routing.module.ts

Hier werden Routen eingestellt – zu Beginn sieht es so aus:

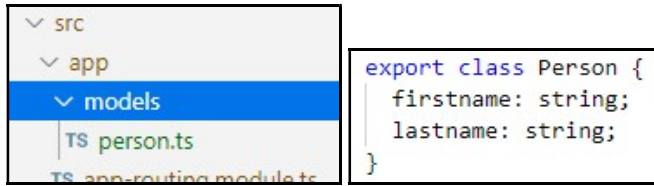
```
const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

2 models

Hierher kommen die Model- bzw. DTO-Klassen.

Neuen Folder **models** manuell erstellen und alle Klassen darin einfügen.



3 Core

Hierher kommen Services, Interceptors und Guards.

3.1 Merkmale

- Enthält die Singletons, die in der gesamten App verwendet werden.
- Üblicherweise **alle Services**
- Üblicherweise **keine Komponenten**
- Werden in app.module.ts importiert - und nur dort

3.2 Erzeugen

Mit **ng generate module Core** wird das Verzeichnis und auch das Modul erzeugt:

```
PS C:\_PR\Angular\ModulesDemo\src\app> ng generate module Core
CREATE src/app/core/core.module.ts (190 bytes)
```

Service erzeugen – dazu ins Verzeichnis core wechseln bzw. **ng g s core\Dummy**:

```
PS C:\_PR\Angular\ModulesDemo\src\app\core> ng g service Dummy
CREATE src/app/core/dummy.service.ts (134 bytes)
```

```
PS C:\_PR\Angular\ModulesDemo\src\app\core> ng g service Other
CREATE src/app/core/other.service.ts (134 bytes)
```

Service macht nichts, außer Output im Konstruktor:

```
@Injectable({
  providedIn: 'root'
})
export class DummyService {
  val = 1;
  constructor() {
    console.log('*** constructor DummyService');
  }
  getNext(): number {
    this.val++;
    return this.val;
  }
}
```

Im Modul dann unter providers alle Services exportieren:

```
@NgModule({
  declarations: [],
  imports: [CommonModule],
  providers: [DummyService, OtherService]
})
export class CoreModule { }
```

Hinweis: Aufgrund von **providedIn: 'root'** müsste man Services nicht mehr in ein Modul packen bzw. bei providers eintragen!

3.3 Verwenden

In app.module.ts wird dann nur noch das gesamte CoreModule importiert, die einzelnen Services jedoch dann nicht mehr explizit:

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    CoreModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

```
*** constructor DummyService
*** constructor OtherService
*** constructor AppComponent
```

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title = 'ModulesDemo';

  constructor(private dummy: DummyService, private other: OtherService) {
    console.log('*** constructor AppComponent');
  }
}
```

4 Shared

Hierher kommen gemeinsame Komponenten und Pipes.

4.1 Merkmale

- Enthält die Komponenten, Direktiven und Pipes die in anderen Modulen verwendet werden.
- Werden in jedem Modul importiert, das Teile davon braucht.

4.2 Erzeugen

Folder und Modul anlegen mit **ng g m Shared**:

```
PS C:\_PR\Angular\ModulesDemo\src\app> ng g m Shared
CREATE src/app/shared/shared.module.ts (192 bytes)
```

In dieses shared-Verzeichnis wechseln → die Komponenten,... werden in shared.module.ts eingetragen.

Komponenten erzeugen (oder wieder **ng g c shared\LabelText**):

```
PS C:\_PR\Angular\ModulesDemo\src\app\shared> ng g c LabelText
CREATE src/app/shared/label-text/label-text.component.html (25 bytes)
CREATE src/app/shared/label-text/label-text.component.ts (285 bytes)
CREATE src/app/shared/label-text/label-text.component.scss (0 bytes)
UPDATE src/app/shared/shared.module.ts (282 bytes)
```

Pipes analog

```
PS C:\_PR\Angular\ModulesDemo\src\app\shared> ng g p InitCaps
CREATE src/app/shared/init-caps.pipe.ts (209 bytes)
UPDATE src/app/shared/shared.module.ts (345 bytes)
```

Evtl. Pipes in ein eigenes Verzeichnis generieren: **ng g p shared\pipes\InitCaps --flat**

4.2.1 shared.module.ts

Komponenten noch von declarations nach **exports** kopieren:

```
@NgModule({
  declarations: [InitCapsPipe, LabelTextComponent],
  imports: [CommonModule],
  exports: [LabelTextComponent, InitCapsPipe]
})
export class SharedModule { }
```

4.3 Verwendung

Wird jetzt nur noch von jenen Modulen importiert, die Teile des Shared-Moduls brauchen.

Falls App.Component z.B. LabelText oder Pipe braucht → bei imports angeben

```
@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    AppRoutingModule,
    CoreModule,
    SharedModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

```
<h1>Angular Modules</h1>

<app-label-text></app-label-text>
<ul>
  <li>{{'THIS should be with initial caps'|initCaps}}</li>
  <li>{{'defg'|initCaps}}</li>
</ul>
```


5 Feature Module eager

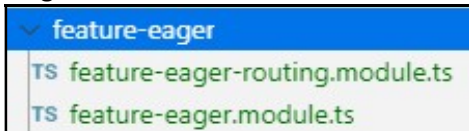
Derartige Module werden gleich zu Beginn geladen, weil sie direkt referenziert werden. Sie enthalten üblicherweise jene Teile der App, die ziemlich sicher vom Benutzer annavigiert werden.

5.1 Erzeugen

Wieder ein Modul im root-Verzeichnis erzeugen, aber mit **--routing**
ng generate module FeatureEager --routing

```
PS C:\_PR\Angular\ModulesDemo\src\app> ng generate module FeatureEager --routing=true
CREATE src/app/feature-eager/feature-eager-routing.module.ts (256 bytes)
CREATE src/app/feature-eager/feature-eager.module.ts (305 bytes)
```

Folgendes entsteht:



Darin jetzt dann Komponenten erzeugen, evtl. mit **--flat**, falls es nicht zu viele sind.

```
PS C:\_PR\Angular\ModulesDemo\src\app\feature-eager> ng g c order-list --flat
CREATE src/app/feature-eager/order-list.component.html (25 bytes)
CREATE src/app/feature-eager/order-list.component.ts (285 bytes)
CREATE src/app/feature-eager/order-list.component.scss (0 bytes)
UPDATE src/app/feature-eager/feature-eager.module.ts (558 bytes)
```

Oder wieder auch: **ng g c feature-eager\OrderList --flat --skipTests**

```
PS C:\_PR\Angular\ModulesDemo\src\app\feature-eager> ng g c order-detail --flat
CREATE src/app/feature-eager/order-detail.component.html (27 bytes)
CREATE src/app/feature-eager/order-detail.component.ts (293 bytes)
CREATE src/app/feature-eager/order-detail.component.scss (0 bytes)
UPDATE src/app/feature-eager/feature-eager.module.ts (477 bytes)
```

5.2 Routing

Obige Seiten sollen über **localhost:4200/orders** bzw. **localhost:4200/orders/123** erreichbar sein. Diese Routen müssen im entsprechenden Routing-Module eingetragen werden.

5.2.1 feature-eager-routing.module.ts

```
const routes: Routes = [
  {
    path: 'orders',
    children: [
      { path: '', component: OrderListComponent },
      { path: ':id', component: OrderDetailComponent }
    ]
  }
];

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

5.2.2 feature-eager.module.ts

Die Referenz auf das RoutingModule der FeatureEager-Komponente wurde automatisch eingetragen. Die declarations ebenfalls:


```
@NgModule({
  declarations: [OrderDetailComponent, OrderListComponent],
  imports: [
    CommonModule,
    FeatureEagerRoutingModule
  ]
})
export class FeatureEagerModule { }
```

5.2.3 app.module.ts

```
@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    CoreModule,
    SharedModule,
    FeatureEagerModule,
    AppRoutingModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Reihenfolge ist wichtig – FeatureEagerModule muss VOR AppRoutingModule importiert werden!

5.3 Verwendung von Core

Services von Core-Modul können automatisch verwendet werden (sind Singletons und wurden schon in App-Modul importiert).

Nicht mehr explizit importieren!

```
export class OrderListComponent implements OnInit {
  orderIds = [11, 22, 33, 44];
  constructor(private dummy: DummyService) {
    console.log('*** constructor OrderListComponent');
    console.log(`  next val: ${dummy.getNext()}`);
  }

  ngOnInit() { }
}
```

5.4 Verwendung von Shared

Wie oben erwähnt, muss dazu das Shared-Modul importiert werden.

Falls man das vergisst, bekommt man eine Fehlermeldung

```
<h1>Orders</h1>
<ul>
  <li *ngFor="let
    <a routerLink=
  </li>
</ul>
```

'app-label-text' is not a known element:

1. If 'app-label-text' is an Angular component, then verify that it is part of this module.
2. If 'app-label-text' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA' to the '@NgModule.schemas' of this component to suppress this message. Angular

Peek Problem No quick fixes available

Shared-Komponent: <app-label-text></app-label-text>

```
► Error: Template parse errors:  
'app-label-text' is not a known element:  
1. If 'app-label-text' is an Angular component, then verify that it is part of this module.  
2. If 'app-label-text' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA' to the  
'@NgModule.schemas' of this component to suppress this message. ("  
  
</ul>  
Shared-Komponent: [ERROR -><app-label-text></app-label-text>"): ng:///FeatureEagerModule  
/OrderListComponent.html@7:18
```

Also in feature-eager.module.ts

```
@NgModule({  
  declarations: [OrderDetailComponent, OrderListComponent],  
  imports: [CommonModule, FeatureEagerRoutingModule, SharedModule]  
})  
export class FeatureEagerModule { }
```

6 Feature Module lazy

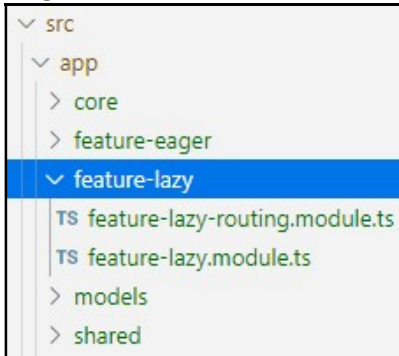
Derartige Module werden erst dann vom Backend geladen, wenn der Benutzer Teile davon aufruft.

6.1 Erzeugung

Zuerst einmal unterscheidet sich die Erzeugung des Moduls nicht von einem Eager Loaded Module.

```
PS C:\_PR\Angular\ModulesDemo\src\app> ng generate module FeatureLazy --routing
CREATE src/app/feature-lazy/feature-lazy-routing.module.ts (255 bytes)
CREATE src/app/feature-lazy/feature-lazy.module.ts (301 bytes)
```

Folgendes entsteht:



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

```
const routes: Routes = [];

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

Darin jetzt wieder die gewünschten Komponenten erzeugen, evtl. auch wieder mit **--flat**, falls es nicht zu viele sind.

```
PS C:\_PR\Angular\ModulesDemo\src\app\feature-lazy> ng g c customer-list --flat
CREATE src/app/feature-lazy/customer-list.component.html (28 bytes)
CREATE src/app/feature-lazy/customer-list.component.ts (297 bytes)
CREATE src/app/feature-lazy/customer-list.component.scss (0 bytes)
UPDATE src/app/feature-lazy/feature-lazy.module.ts (389 bytes)
```

```
PS C:\_PR\Angular\ModulesDemo\src\app\feature-lazy> ng g c customer --flat
CREATE src/app/feature-lazy/customer.component.html (23 bytes)
CREATE src/app/feature-lazy/customer.component.ts (278 bytes)
CREATE src/app/feature-lazy/customer.component.scss (0 bytes)
UPDATE src/app/feature-lazy/feature-lazy.module.ts (466 bytes)
```

6.2 Routing

6.2.1 feature-lazy-routing.module.ts

```
const routes: Routes = [  
  {  
    path: '',  
    children: [  
      { path: '', component: CustomerListComponent },  
      { path: ':name', component: CustomerComponent }  
    ]  
  }  
];  
  
@NgModule({  
  imports: [RouterModule.forChild(routes)],  
  exports: [RouterModule]  
})  
export class FeatureLazyRoutingModule { }
```

Man beachte, dass bei path ein Leerstring eingetragen ist – der Subpath „customers“ wird in app-routing.module.ts eingetragen.

6.2.2 feature-lazy.module.ts

Hier ist kein Unterschied zu einem eager loaded module.

```
@NgModule({  
  declarations: [CustomerListComponent, CustomerComponent],  
  imports: [CommonModule, FeatureLazyRoutingModule]  
})  
export class FeatureLazyModule { }
```

6.2.3 app.module.ts

Dieses Modul wird **NICHT** in AppModule importiert. Grund: damit hätte man eine Referenz und es würde automatisch als Abhängigkeit beim Start mitgeladen.

6.2.4 app-routing.module

Hier wird der Subpath „customers“ angegeben, anstelle von „component“ wird „**loadChildren**“ notiert, wobei die tatsächliche Modulklassse als Lambda-Ausdruck angegeben und somit asynchron geladen wird.

```
const routes: Routes = [  
  {  
    path: 'customers',  
    loadChildren: () => import('./feature-lazy/feature-lazy.module').then(x => x.FeatureLazyModule)  
  }  
];  
  
@NgModule({  
  imports: [RouterModule.forRoot(routes)],  
  exports: [RouterModule]  
})  
export class AppRoutingModule { }
```

In der Developerkonsole sieht man auch, dass das entsprechende Javascript-File erst dann geladen wird, wenn man zum ersten Mal auf Customers navigiert.

Status	Met...	Host	Datei	Ursprung	Typ
304	GET	localhost:4...	feature-lazy-feature-lazy-module.js	script	js

Auch am Compile-Vorgang sieht man, dass ein neuer Chunk entsteht:

```
3 unchanged chunks
chunk {feature-lazy-feature-lazy-module} feature-lazy-feature-lazy-module.js, fea
chunk {main} main.js, main.js.map (main) 46.4 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 9.03 kB [entry] [rendered]
Time: 424ms
@wdm: Compiled successfully.
```

6.3 Verwendung von Core

Es gilt dasselbe wie für eager loaded modules.

Services von Core-Modul können automatisch verwendet werden (sind Singletons und wurden schon in App-Modul importiert).

Nicht mehr explizit importieren!

```
export class CustomerListComponent implements OnInit {
  customerNames = ['Hansi', 'Susi', 'Pepi'];
  constructor(private dummy: DummyService) {
    console.log('*** constructor CustomerListComponent');
    console.log(`    next val: ${dummy.getNext()}`);
  }

  ngOnInit() { }
}
```

6.4 Verwendung von Shared

Auch hier gilt dasselbe wie für eager loaded modules.

Wie oben erwähnt, muss dazu das Shared-Modul importiert werden.

Falls man das vergisst, bekommt man eine Fehlermeldung

```
<h1>Customers</h1>
<ul>
  <li *ngFor="let name of customerNames">
    <a routerLink='{{name}}'>Customer {{name}}</a>
  </li>
</ul>
Shared-Komponent: <app-label-text></app-label-text>
```

Also in feature-eager.module.ts

```
@NgModule({
  declarations: [CustomerListComponent, CustomerComponent],
  imports: [CommonModule, FeatureLazyRoutingModule, SharedModule]
})
export class FeatureLazyModule { }
```

6.5 Eigenes Service

In einem Lazy Feature Module kann man auch Services definieren, die nur von diesem Feature benutzt werden.

```
PS C:\_PR\Angular\ModulesDemo\src\app\feature-lazy> ng g s Customer
CREATE src/app/feature-lazy/customer.service.ts (137 bytes)
```

Es wird im zugehörigen Modul (also feature-lazy.module.ts) bei providers angegeben:


```
@NgModule({
  declarations: [CustomerListComponent, CustomerComponent],
  imports: [CommonModule, FeatureLazyRoutingModule, SharedModule],
  providers: [CustomerService]
})
export class FeatureLazyModule { }
```

Dabei sollte man **providedIn: root** nicht angeben:

```
@Injectable()
export class CustomerService {

  constructor() {
    console.log('*** constructor CustomerService');
  }
}
```

Die Verwendung ist aber wie jedes andere Service auch, also mit Dependency Injection:

```
export class CustomerListComponent implements OnInit {
  customerNames = ['Hansi', 'Susi', 'Pepi'];
  constructor(private dummy: DummyService, private customer: CustomerService) {
    console.log('*** constructor CustomerListComponent');
    console.log(`    next val: ${dummy.getNext()}`);
  }

  ngOnInit() { }
}
```

7 Vergleich eager/lazy

Eager und Lazy Module unterscheiden sich zusammengefasst dadurch:

Datei	Eager	Lazy
app.module.ts	Modul bei imports angeben	Kein Eintrag
{modulename}- routing.module.ts	Name der Subroute	"
app-routing.module.ts	Kein Eintrag	<pre>Pfad in Array Routes[] { path: 'subroute', loadChildren: () => import('./{modulename}/{modulename}.module') .then(x=>x.{moduleclassname}) }</pre>

8 Zusammenfassung

8.1 Dateistruktur

Die Dateistruktur sieht so aus:

```

└─ app
   ├── core
   ├── feature-eager
   ├── feature-lazy
   ├── models
   ├── shared
   ├── app-routing.module.ts
   ├── app.component.html
   ├── app.component.scss
   ├── app.component.ts
   ├── app.module.ts
   └── preload-strategy-custom.ts
```

Man hat also pro Feature ein eigenes Modul, wobei man selbst entscheidet, ob das lazy oder eager geladen werden soll.

8.2 Moduleigenschaften

Siehe <https://angular.io/guide/module-types>

Module	Declarations	Providers	Exports	Imported by
Domain	Yes	Rare	Top component	Feature, AppModule
Routed	Yes	Rare	No	None if lazy
Service/Core	No	Yes	No	AppModule
Widget/Shared	Yes	Rare	Yes	Feature

9 Preload Strategies

Lazy loaded modules werden ja erst bei Bedarf nachgeladen.

Man kann dieses Laden jedoch beeinflussen, indem man z.B. bestimmte Lazy Module trotzdem schon explizit nach dem Start lädt.

9.1 Preload all

Möchte man alle Module zu Beginn laden, gibt man folgendes an:

```
@NgModule({
  imports: [RouterModule.forRoot(routes, { preloadingStrategy: PreloadAllModules })],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Dadurch werden alle Module geladen und somit auch das Customer-Modul, wie man in der Netzwerkanalyse der Developerkonsole erkennen kann.

9.2 custom strategy

Dazu muss man eine Klasse erzeugen, die das Interface PreloadingStrategy implementiert, z.B.:

```
export class PreloadStrategyCustom implements PreloadingStrategy {
  preload(route: Route, load: () => Observable<any>): Observable<any> {
    return route.data && route.data.preload ? load() : of(null);
  }
}
```

In app-routing.module.ts kann man dann für jede lazy route angeben, ob diese automatisch geladen wird, indem man die Property preload (die ja oben abgefragt wird) entsprechend setzt:






```
const routes: Routes = [
  {
    path: 'customers',
    loadChildren: './feature-lazy/feature-lazy.module#FeatureLazyModule',
    data: { preload: true }
  }
];

You, a few seconds ago | 1 author (You)
@NgModule({
  imports: [RouterModule.forRoot(routes, { preloadingStrategy: PreloadStrategyCustom })],
  exports: [RouterModule],
  providers: [PreloadStrategyCustom]
})
export class AppRoutingModule { }
```

Mit preload:true sieht der Start so aus:

200	GET	localhost:4... polyfills.js	script	js
200	GET	localhost:4... styles.js	script	js
200	GET	localhost:4... vendor.js	script	js
200	GET	localhost:4... main.js	script	js
200	GET	localhost:4... feature-lazy-feature-lazy-module.js	script	js

Mit preload:false dann so:

200	GET	 localhost:4... runtime.js	script	js
200	GET	 localhost:4... polyfills.js	script	js
200	GET	 localhost:4... styles.js	script	js
200	GET	 localhost:4... vendor.js	script	js
200	GET	 localhost:4... main.js	script	js