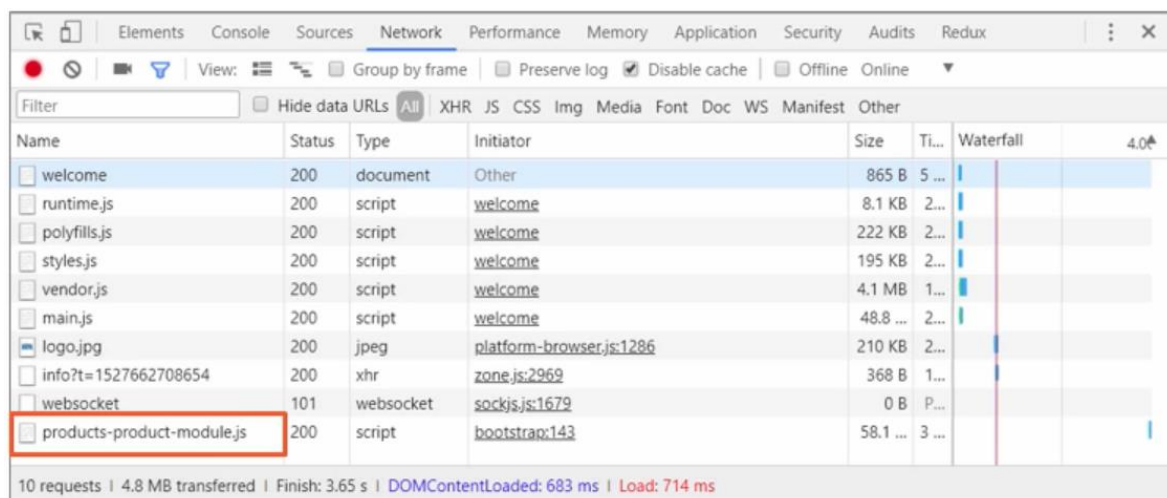


## Tipado fuerte del estado – Extender la interfaz de estado para módulos con lazy loading

Cuando configuramos un *feature module* para *lazy loading*, ese módulo se compila de forma independiente:

```
chunk {main} main.js, main.js.map (main) 59 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 141 kB [initial] [rendered]
chunk {products-product-module} products-product-module.js, products-product-module.js.map (products-product-module) 51.8 kB [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 9.02 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 684 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 3.62 MB [initial] [rendered]
```

Cuando el usuario accede a la aplicación, este módulo configurado con *lazy loading* se descarga del servidor separado de nuestro *bundle* principal después de que se muestre la primera página de la aplicación:



The screenshot shows the Chrome DevTools Network tab with the 'Network' panel selected. The 'Filter' is set to 'All'. The table lists the following resources:

Name	Status	Type	Initiator	Size	Ti...	Waterfall
welcome	200	document	Other	865 B	5 ...	
runtime.js	200	script	welcome	8.1 KB	2...	
polyfills.js	200	script	welcome	222 KB	2...	
styles.js	200	script	welcome	195 KB	2...	
vendor.js	200	script	welcome	4.1 MB	1...	
main.js	200	script	welcome	48.8 ...	2...	
logo.jpg	200	jpeg	platform-browser.js:1286	210 KB	2...	
info?t=1527662708654	200	xhr	zone.js:2969	368 B	1...	
websocket	101	websocket	sockjs.js:1679	0 B	P...	
products-product-module.js	200	script	bootstrap:143	58.1 ...	3 ...	

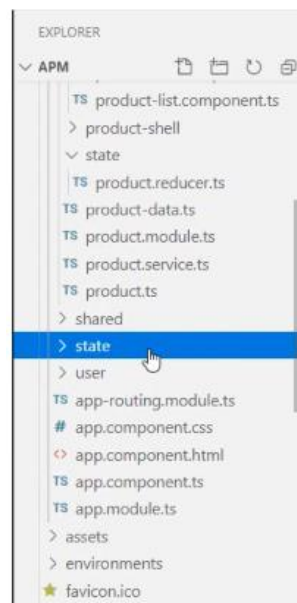
At the bottom, the summary shows: 10 requests | 4.8 MB transferred | Finish: 3.65 s | DOMContentLoaded: 683 ms | Load: 714 ms. The 'products-product-module.js' resource is highlighted with a red box in the original image.

Esto mejora el rendimiento de inicio de nuestra aplicación.

¿Qué tiene que ver el *lazy loading* con nuestra interfaz *State*?

Queremos establecer límites lógicos alrededor de nuestros features con *lazy loading*. Para mantener ese límite, queremos mantener nuestros features completamente separados de nuestro módulo principal.

Recordemos que la interfaz *State* la colocamos al nivel de la carpeta raíz de nuestro proyecto (carpeta *app*), eso significa que la interfaz *State* se descargará a nuestro navegador dentro del *bundle* principal de nuestra aplicación:



Al importar directamente la interfaz *ProductState* dentro de nuestro archivo de la interfaz *State*, rompemos ese límite y el *lazy loading*:

```
app
import { ProductState } from '../products/state/product.reducer';
import { UserState } from '../user/state/user.reducer';

export interface State {
  products: ProductState;
  users: UserState;
}
```

La descarga por *lazy loading* se rompe ya que, en nuestro módulo principal de la aplicación, estamos importando una interfaz que tenemos declarada en otro módulo que descargamos con *lazy loading*, en el módulo *products*.

En su lugar, definimos la interfaz sólo para el *slice* de estado “*user*”, ya que éste no se descarga con lazy loading:

app

```
import { UserState } from '../user/state/user.reducer';

export interface State {
  users: UserState;
}
```

Ahora, en nuestra interfaz *State*, que está definida dentro de nuestro módulo principal de la aplicación, ya no tenemos importada la interfaz *ProductState*.

A continuación, en nuestro código del *feature product*, extenderemos la definición de la interfaz *State* para incluir la interfaz *ProductState*:

products

```
import * as AppState from '../../state/app.state';

export interface State extends AppState.State {
  products: ProductState;
}
```

Dado que este código forma parte del módulo *feature product*, lo mantenemos dentro de los límites de descarga de *lazy loading*. Lo que hacemos aquí, es definir otra interfaz *State* que extiende de nuestra interfaz *State* del módulo principal de la aplicación, lo que provoca que la interfaz se sobrescriba con esta nueva que estamos definiendo:

app

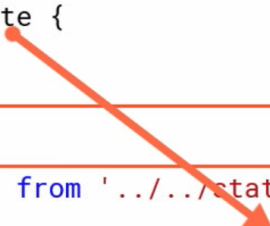
```
import { UserState } from '../user/state/user.reducer';

export interface State {
  users: UserState;
}
```

products

```
import * as AppState from '../../state/app.state';

export interface State extends AppState.State {
  products: ProductState;
}
```



Extendemos el estado global de la aplicación con una interfaz *State* que se parece a la original, pero definida para mantener intactos los límites de descarga del *lazy loading*:

#### app

```
import { ProductState } from '../products/state/product.reducer';  
import { UserState } from '../user/state/user.reducer';
```

```
export interface State {  
  products: ProductState;  
  users: UserState;  
}
```

#### app

```
import { UserState } from '../user/state/user.reducer';
```

```
export interface State {  
  users: UserState;  
}
```

#### products

```
import * as AppState from '../../state/app.state';
```

```
export interface State extends AppState.State {  
  products: ProductState;  
}
```

A medida que añadimos más features, si éstos no están configurados con *lazy loading*, los añadiremos en la interfaz *State* principal:

#### app

```
import { UserState } from '../user/state/user.reducer';
```

```
export interface State {  
  users: UserState;  
}
```

De lo contrario, utilizaremos esta otra técnica en el feature correspondiente para extender la interfaz *State*:

#### products

```
import * as AppState from '../../state/app.state';
```

```
export interface State extends AppState.State {  
  products: ProductState;  
}
```

Volvamos al código y arreglemos nuestra interfaz `State` para mantener los límites del *lazy loading*.

En el archivo `app.state.ts`, eliminamos el atributo `product` de nuestra interfaz `State` junto con la declaración `import` correspondiente:

```
TS product.reducer.ts TS app.state.ts X
1 import { ProductState } from "../products/state/product.reducer";
2
3 export interface State {
4   products: ProductState;
5   user: any;
6 }
7
```

```
TS product.reducer.ts TS app.state.ts X
1 export interface State {
2   user: any;
3 }
4
```

En el archivo `product.reducer.ts`, añadiremos una declaración `import` para acceder al estado global de nuestra aplicación, y extendremos la interfaz `State` para incluir el *slice* de estado `"products"`:

```
TS product.reducer.ts X TS app.state.ts
1 import { createReducer, on, createAction } from '@ngrx/store';
2 import * as AppState from '../../state/app.state';
3
4 import { Product } from '../product';
5
6 export interface State extends AppState.State {
7   products: ProductState;
8 }
9
10 export interface ProductState {
11   showProductCode: boolean;
12   currentProduct: Product;
13   products: Product[];
14 }
15
16 export const productReducer = createReducer(
17   { showProductCode: true },
18   on(createAction('[Product] Toggle Product Code'), state => {
19     return {
20       ...state,
21       showProductCode: !state.showProductCode,
22       myFavoriteMovie: 'LOTR'
23     };
24   })
25 );
26
```

Ahora tenemos una única interfaz `State` que representa todo nuestro árbol de estado. A continuación, utilizaremos estas interfaces para tipear fuertemente nuestro estado.