Summary

Which common pain points and questions do I tackle in this video and article?

1. What's up with all these Angular versions? (0:31)
2. Do you really need the Angular CLI? (4:33)
3. Which Visual Studio Code Extensions can I recommend? (7:45)
4. How can you debug Angular apps? (8:37)
5. How can you pass data from A to B (e.g. between components) in Angular apps? (14:44)
6. "Can I use Angular with PHP/Node/..."? (16:53)
7. "Can I use Angular with Redux"? (18:29)
8. How to prevent state loss after page refreshes? (19:51)
9. "Can I host my Angular app on Heroku"? (22:46)
10. How to fix broken routing after deployment (24:24)
11. Can everyone see your code? (26:14)
12. How to integrate 3rd party CSS and JS packages (27:05)

What's up with all these Angular versions?

When starting off with Angular, it can be really confusing. There are a lot of different versions. So what's up with

- Angular 1
- Angular 2
- Angular 3
- Angular 4
- ...?

It can be confusing but it's actually quite simple: Angular 1 was released in 2010 and it is a complete different framework than Angular 2+. The **+** is important - Angular 2 was a complete re-write of Angular 1 and Angular 3, 4, 5 are simply the latest versions of Angular 2 (Angular 3 actually never came out due to versioning conflicts).

Yes, this sounds strange.

The reason for this strange versioning is that Angular adopted semantic versioning at the end of 2016 (see this and this blog post).

So whilst Angular 5 sounds like a totally different framework, it actually isn't. The syntax hasn't changed since then (and is not about to change over the next released either). There have been some changes, most of them happened behind the scenes though.

Important: We nowadays refer to Angular 2+ as just **Angular** whereas Angular 1 is now called **AngularJS**. So if you read "Angular", people typically mean Angular 2 or later.

So if you learn Angular 5/6/7 know, you'll be prepared for Angular 6/7/8, too.

Do you really need the Angular CLI?

This is another question I see a lot: "Do I really need the Angular CLI ?".

The answer is: Yes! You **should** really need it because Angular requires a more complex build workflow.

Why is such a build workflow required? Because Angular uses TypeScript which needs to be compiled to JavaScript. Because Angular should compiled its template code before it gets shipped to production to improve performance. And because Angular apps should be optimized (e.g. unnecessary code should be removed) before the app gets shipped.

gets shipped.

All these things are taken care of by the CLI and that's why you should really use it. It makes your life as a developer easier and your apps better. And if you ever need full access to the underlying Webpack config, you can always eject from the CLI-managed setup via `ng eject`.

Connected to that, I sometimes get the questions why you need to install Node.js to developer Angular apps. It's a valid question - we're not writing any Node.js code after all! But Node.js is required for two reasons:

1. It ships with npm - the **N**ode **P**ackage **M**anager. This tool is the de-facto standard to manage dependencies of frontend and backend projects. Dependencies include production dependencies like the Angular package itself (which you therefore don't have to download or include from a CDN manually) but also packages like Webpack - a tool which is used by the CLI to bundle all your code together and optimize it
2. It powers the build workflow, i.e. it executes a bunch of scripts that compile and optimize your code.

Which Visual Studio Code Extensions can I recommend?

Visual Studio Code has become a really great IDE for developing Angular apps, I can only recommend using it, especially since it's free.

One of the major advantages of VSC is its extensibility. You can add dozens of extensions to tailor the editor to your needs. But which extensions do you really need?

The good thing is: For Angular development, you really need only one extension! Angular Essentials by John Papa! It bundles a lot of other useful extensions together, so you actually get more than one extension.

How can you debug Angular apps?

Debugging Angular apps is something a lot of people struggle with. Let

Debugging Angular apps is something a lot of people struggle with. Let me share the most important tools and tricks to debug your app

effectively!

Read the error messages carefully

Seriously, this sounds trivial but a lot of people don't do it! You should really read those messages, they do help you and whilst some of them can be cryptic, most of them are quite understandable and helpful!

```
Uncaught Error: Template parse errors:     webpack-internal:///…5/compiler.js:24547
'app-product' is not a known element:
1. If 'app-product' is an Angular component, then verify that it is part of this
module.
2. If 'app-product' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA' to the
'@NgModule.schemas' of this component to suppress this message. ("<h1>My awesome
Shop</h1>
[ERROR ->]<app-product></app-product>
"): ng:///AppModule/AppComponent.html@1:0
```

This message is actually quite clear. It doesn't understand the `app-product` selector you're using somewhere. This is a good starting point to dive deeper and explore what could be the error. Some common error sources would be:

- You have a typo in the selector in `@Component()`
- You didn't add the component to the `declarations[]` array in your `AppModule`

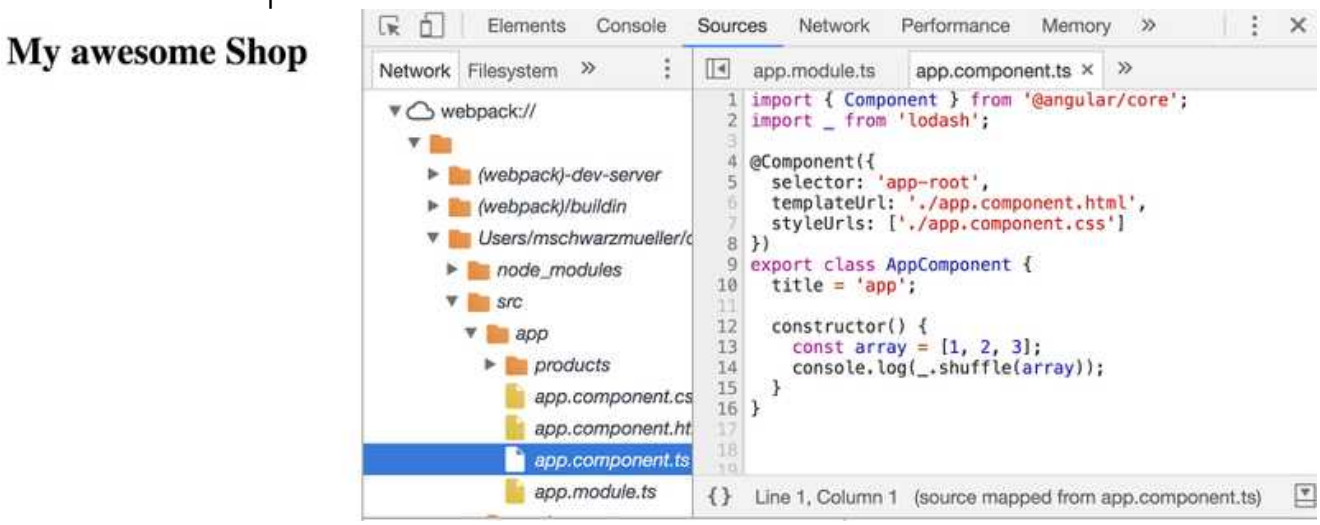
Even if you don't immediately have these two error sources in mind, you should be able to eventually find them. After all the error is that Angular doesn't know this selector, so you should look at all places where this selector and component plays an important role.

Use `console.log()` to get quick insights into your values

This is kind of a "dirty" trick but it can be really useful to put some `console.log()` statements into your code. Why? Because this allows you to quickly look at the value of some variable or property in some execution step in your app. It's done quickly and can give you the hint you just needed to fix some problem.

Use the browser developer tools

Using `console.log()` is alright for some quick and dirty debugging but a better way is to use the browser developer tools every major browser ships with. Here's a detailed guide to the Chrome Developer Tools for example.



In these tools, you can directly access your source code (the JavaScript code), place breakpoints, analyze values of variables and properties and do much more. The one problem you'll eventually face is that your shipped source code doesn't look like the code you wrote though. It's not written in TypeScript, it was compiled to JavaScript.

The good thing is: The CLI setup gives you sourcemaps - little translation helpers that allow the browser to map the compiled JS files back to the original source files. Sounds like magic? It's really useful! It allows you to actually access your original TypeScript source code in the developer tools so that you can place breakpoints and analyze your values directly in that TypeScript code.

In Chrome, you should look for a **webpack://** folder in the **Sources** tab of your dev tools. You should find the TypeScript code there!

Use Augury

Augury is an extension for your developer tools that was built to make the debugging of Angular apps easier. You install it as a Chrome extension and can then use it from inside the Chrome dev tools.

With Augury, you can have a look at your component tree (as it's currently rendered to the screen), the state of all your components in that tree. injected values and much more. It's an extremely useful

addition that allows you to get even deeper insights into Angular.

If you're using NgRx: Use the Redux Devtools!

If you're using NgRx in your Angular app, you should also use the Redux Devtools Extension. Together with the NgRx Devtools, you get detailed insights into the current state of your Redux/ NgRx store, the actions you dispatched and much more.

Extremely useful!

How can you pass data from A to B (e.g. between components)?

Passing data between components, components and services and components and directives is a common problem in Angular apps. We got a couple of different tools to make sure every piece of our Angular app knows what it needs to know but which tools are this exactly? How and when do you use which "tool"?

Tool 1: Property and Event Binding

Property and event binding are core concepts of Angular apps! **Property binding** simply means that you pass data **into** some other element - that could be a native HTML/DOM element or your custom component. So you can set a property of that HTML/DOM element or your component from outside.

Here's an example:

```
@Component({...})
export class LoadedProductComponent {
  @Input() loadedProduct: Product;
}

@Component({
  template: `<app-loaded-product [loadedProduct]="products[0]"></app-loaded-proc
})
export class ProductsComponent {
  products: Product[] = [{name: 'Milk', amount: 2}]
```

```
}
```

In this snippet, the `LoadedProductComponent` has a bindable property `loadedProduct`. It is bindable (that means: settable from outside) because it's annotated with the `@Input()` decorator.

`ProductsComponent` uses this feature and passes data into the `LoadedProductComponent`: The first element (index `0`) of its (non-bindable!) `products` array.

Event binding works exactly the other way around. It allows you to emit your own events, which of course gives you a way of information the parent component of another component about some change and optionally pass data with the event. Just as the native `click` event also gives you an object with informations about the event (e.g. the `event.target`) by default.

It works like this:

```
@Component({...})
export class LoadedProductComponent {
  @Output() productOrdered = new EventEmitter(this.orderedProduct);

  productClicked(selectedProduct) {
    this.productOrdered.emit(selectedProduct);
  }
}

@Component({
  template: `<app-loaded-product (productOrdered)="doSomething($event)"></app-l
})
export class ProductsComponent {
  doSomething(orderedProduct) { ... }
}
```

Here, the `LoadedProductComponent` emits a custom event `productOrdered` - created with EventEmitter, which ships with Angular - some `selectedProduct` whenever `productClicked` is executed (that could be happening because maybe a listener to the

native `click` event was added to some DOM element).

The event is **listenable** from outside (i.e. from the parent component - `ProductsComponent` in this case) because `@Output()` was added as a decorator to `productOrdered`.

In the `ProductsComponent`, a listener for `productOrdered` is added to `<app-loaded-product>` and the `doSomething()` method is executed whenever the event occurs. The `$event` variable which is passed as an argument is a special variable exposed by Angular that simply carries the event payload (i.e. the emitted product - `selectedProduct`).

Tool 2: Services

Property and event binding is extremely useful but you can only pass data from parent to child component and the other way around. What if you need to pass data from a grandchild to some other component? So what do you do if no direct connection between two components exists?

You can follow two routes:

- You can build a long chain of `@Input()` s and `@Output()` s to get data from A to B. This can be a good solution to build highly re-useable components that can be dumped into the template anywhere and only require certain input/ offer certain output.
- But in many cases, you might want to look at **Services**

Services are simply normal TypeScript classes that can be injected by Angular. Injection means that Angular instantiates the class and provides that instance to a component via its selectors. You can learn more about Services and Dependency Injection here .

Since services can be injected into any component (and also into other services or directives), you can use them to centralize data and access it from anywhere in your application. Really convenient!

To pass data around efficiently, you typically use RxJS subjects as event emitters like this:

```
// in data.service.ts
import { Subject } from 'rxjs/Subject';

export class DataService {
  dataAdded = new Subject<any>();

  addData(newData) {
    this.dataAdded.next(newData);
  }
}

// in data-receiving.component.ts
import { Subscription } from 'rxjs/Subscription';
import { DataService } from './path/to/data.service'

@Component({...})
export class DataReceivingComponent implements OnInit, OnDestroy {
  dataSub: Subscription;

  constructor (private dataService: DataService) {}

  ngOnInit() {
    this.dataSub = this.dataService.dataAdded.subscribe(data => {
      // do something with the data
    });
  }

  ngOnDestroy() {
    if (this.dataSub) {
      this.dataSub.unsubscribe(); // required to prevent memory leaks
    }
  }
}
```

Tool 3: NgRx (Redux)

For bigger apps, you may still end up with a complex construct of services that get injected into different places and have challenging relations. A way to introduce a clear pattern of data flow into your app is

to use Redux .

Redux simply is a package that offers you a certain collection of tools and enforces a clear pattern of using these tools to make state management simply. Put simply, you have a central store where you have all your app state (e.g. some selected product) and you can access that store from your components etc. You don't have dozens of services communicating with each other then.

For Angular, you typically don't use the Redux package itself (though you could do that, it's not limited to be used in React apps!) but you use NgRx - an Angular-specific Redux implementation that follows the same logic.

"Can I use Angular with PHP/Node/...?"

I often see the question whether you can use Angular with PHP, Node.js or some other server-side language.

And the answer is: **Yes**, absolutely!

Angular doesn't care about your backend and your server-side language, it's a client-side framework, it runs in the browser!

With Angular, you build Single-Page-Applications (SPAs) and these aren't rendered on the server. They only communicate with servers through Ajax requests (via Angular's built-in HttpClient), hence your backend needs to provide a (RESTful) API to which Angular can send its requests. And that's all!

One exception is important though: If you're using Angular Universal , you'll still not use Angular to write server-side code (i.e. to access databases, work with file storage or anything like that) but you can **pre-render** your Angular apps on the server. That only works with Node.js as of now though, so if that's important to you, you should go with Node.js

“Can I use Angular with Redux?”

Yes! This question was already kind of answered in the Pass data from A to B point. You can use the Angular-specific Redux implementation NgRx to implement a central store and other redux features into your Angular app.

How to prevent State Loss after Page Refreshes

Here’s a question I observed a lot: “If I press the refresh button in my browser, my app state resets. How can I prevent this?”

It’s true! If you hit that refresh button, your page reloads and since Angular runs entirely in JavaScript, your script restarts and you app state is lost. That’s not a bug but the expected behavior.

If you’re coming from a “traditional” web development environment, where you built web pages by rendering views in your server-side language, that’s a strange behavior. You used to use **Sessions** to manage the user state for example. Page refreshes would therefore not lose that state.

In single page applications as Angular (or other frameworks and libraries) creates them, you don’t use sessions though. The reason is simple: Your app is decoupled from your backend. You only have one single page in the end and your (RESTful) API to which you talk occasionally doesn’t care about your app - it’s stateless.

So you don’t use sessions - what do you do instead?

You have two common options to still persist user state (e.g. “Is the user logged in”?):

1. Use localStorage to store simple key-value pairs like a JSON Web Token (JWT) you got from your backend (a pattern commonly used for authentication in SPAs).
2. Use a server-side database for state/ data that needs to persists long-term

long-term.

localStorage is a common choice to store client state. It's lost if the user clears all browser data, so it's not suitable for data that needs to be stored long-term. It's also accessible via JavaScript (and by the user), so you shouldn't store security-relevant information there. You do regularly use it to store JWTs though - tokens that are short-lived (for security reasons) and grant the user access to some protected resources on the backend.

You can use localStorage to store state across page refreshes by following this pattern:

1. Store data in localStorage once you have it (e.g. store a JWT once you received it).
2. When your app starts, check if the data is stored in localStorage - a good place is inside the `ngOnInit` lifecycle method in your `AppComponent` since that will execute right at the start of your application life.
3. Initialize your app with any (optionally validated) data you got from localStorage if available.

By using this pattern, you can start your app in the same state you left it.

Use server-side databases for any data that's not really relevant to the client upon page refreshes but which should instead be stored long-term or which should be synchronized across different devices (e.g. a selected user location).

"Can I host my App on Heroku etc?"

"Can I host my Angular app on Heroku or a comparable service"?

That's a question I also see a lot!

The answer is: Generally, yes - you can of course host your app on any web hosting service. But there are services which are better suited than others. Heroku for example is a service that's built to allow you to easily

others. Heroku for example is a service that's built to allow you to easily host PHP, Node.js etc. apps. It spins up fitting environments (i.e.

configures the servers, interpreters etc needed by the selected language) and makes sure that you don't have to spend hours setting this all up.

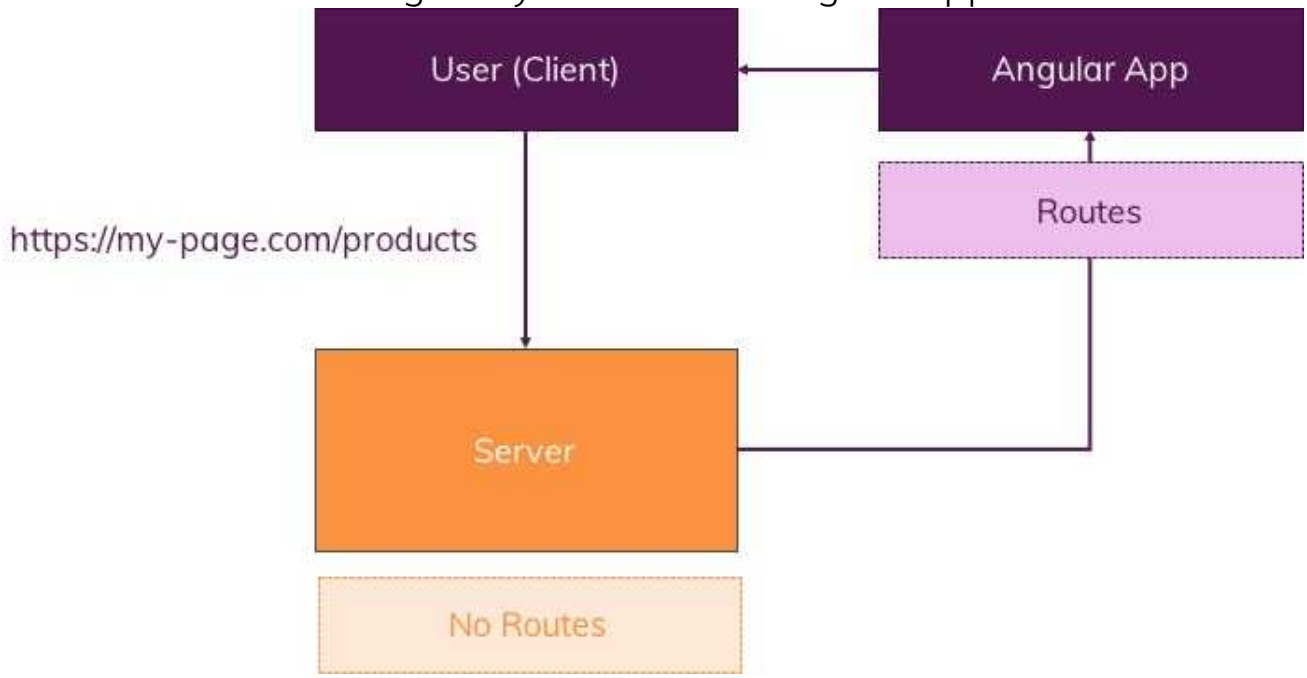
But Angular doesn't use any server-side language! It's - after your ran `ng build --prod` just a bunch of JavaScript and CSS files as well as the `index.html` file. You don't need a Node server for that!

Therefore, for Angular apps, static website hosts like AWS S3 or Firebase Hosting are better choices. They're typically cheaper, super-easy to setup and don't require you to add overhead code (like a dummy Node.js server) to just ship your files to your users.

How to fix broken Routes after Deployment

After deploying an Angular app to a real server, I sometimes hear that the routing stops working. At least if users directly visit any other page than the main page - say `https://my-app.com/products` or if users refresh the browser whilst being on such a "sub-page".

To understand this "error" (it's not really an error), we have to understand how routing really works in an Angular app.



Angular stores and manages your routes, **not** the server that's hosting your Angular app! Keep in mind that that server only serves the

`index.html` file - that's about its only job!

Therefore, your server can't do anything with an incoming request pointing at a `/products` route - it's totally unknown to it! Due to the way the web works, requests reach the server though and your client (i.e. the Angular app) doesn't even get loaded as the server throws a **404 error**.

So what can we do to solve that issue?

The solution is simple: Your server should **always** serve the `index.html` file, especially in 404 error cases! If you configure your server to do that (static hosts like AWS S3 or Firebase provide easy-to-use configurations to achieve this behavior), your Angular app gets loaded and gets a chance to handle the request.

If you still want to render a 404 error page for routes that are really unknown, i.e. that are also not configured in your Angular app, you'll ne to add a **catch-all** route like this:

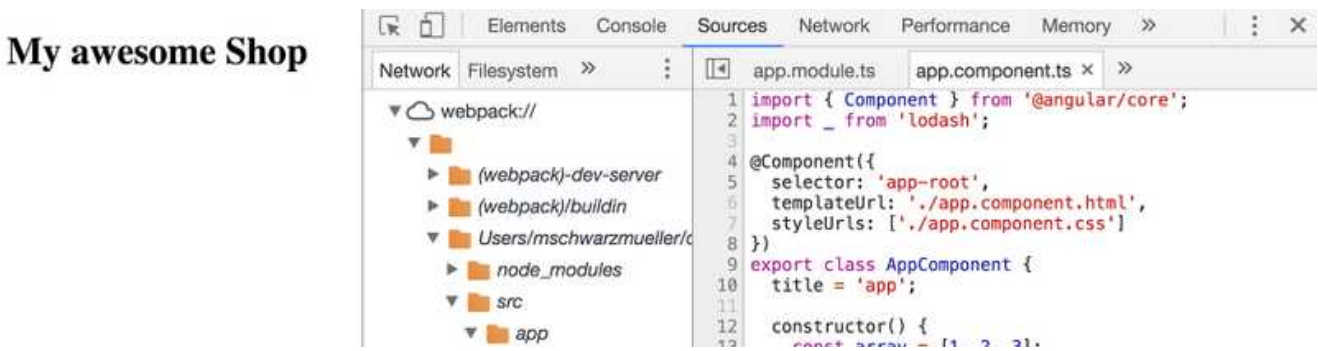
```
// Configure all other routes first or they'll not be considered!
{ path: '**', component: Custom404PageComponent }
```

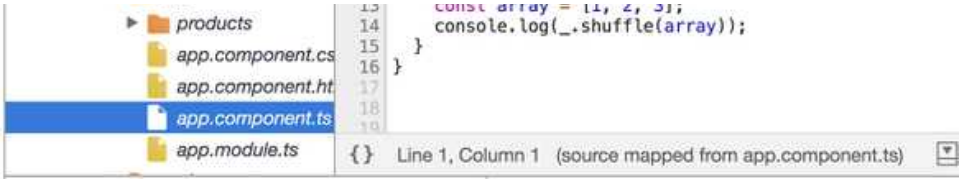
Can Everyone see your Code?

I explained how you can debug your Angular app.

If you can look into your JS (and even TypeScript) code - can't everyone else do the same?

The answer is: Yes!





JavaScript runs in the browser and it's **not pre-compiled** - this means that it's readable by everyone who's on your web page. It may be minified and obfuscated but this can only make it harder, not prevent it. There's nothing you can do about that!

Is this a deal breaker though?

No! It just means that you shouldn't put security-relevant or sensible information into your frontend JavaScript code. All your other code may be accessible but it typically also doesn't hold any important information for hackers.

Due to your code being accessible to everyone, you should always rely on server-side validation of user input though. You can write the best validation code in the world using Angular's form validation capabilities but since any use can simply re-write or break that code, you always need to validate on the server, too!

Frontend code is all about providing a good user experience (e.g. via instant validation results) **not** about securing your database.

How to integrate 3rd Party CSS and JS Libraries

"Can I use jQuery in my Angular app" and "how can I use Bootstrap in my Angular app" are questions I see a lot.

And it makes sense - writing Angular apps looks like you're working with a totally different language! It can be hard to understand that you're still writing a frontend, JavaScript-based app in the end!

But since you're doing that, integrating third-party libraries is actually rather easy!

Integrating 3rd Party JavaScript Libraries

Let's have a look at third-party JavaScript libraries like jQuery first as this

is a bit more “special”.

Libraries with `.d.ts` Files

We first of all have to differentiate between libraries that ship with type definition files (`.d.ts` files) even though the library itself might be written in JavaScript. Quite a lot of libraries do that these days. TypeScript can understand such libraries once you start importing objects, methods etc. from them.

Let’s have a look at an example: The Firebase SDK for JavaScript.

You can add it to your project via

```
npm install --save firebase
```

Once you added it, you can use it in your files like this:

```
import * as firebase from 'firebase'
```

We’re essentially importing from that library as we import from other (custom created) files. We’ll get autocompletion (if supported by the IDE) and TypeScript support because Firebase actually ships a `.d.ts` file with the SDK.

What do these `.d.ts` files do then?

They act as “translation helpers” - they allow TypeScript to understand the structure and types of the JavaScript library, even though it’s not written with TypeScript.

Libraries with ES6 exports but without `.d.ts` Files

We also have libraries that do use ES6 exports (i.e. we can import with the ES6/ TypeScript `import { something } from 'library'` statements) but don’t ship a `.d.ts` files.

Whilst we won’t get TypeScript support in this case, we should still be

able to import by using the above mentioned syntax.

Libraries without ES6 exports and `.d.ts` Files

We also have libraries that are not offering any ES6 exports or `.d.ts` files. Often, these are JS packages which you include via a CDN or simply download - so you didn't use `npm install` to fetch them.

In such cases, have three options of including them:

- Add a `<script src="path/to/library">` tag in the `index.html` file
- Add the path (relative from the `src/` directory!) to the `scripts[]` array in the `.angular-cli.json` file
- Add an import to the `main.ts` file in your Angular project: `import 'path/to/file`

The latter two options will lead to the library file being included in your final bundle and hence you can access anything the library offers in your TypeScript files.

Here's how you could install + add Lodash :

```
npm install --save lodash

import 'lodash';

@Component({...})
export class MyComponent {...}
```

One important note though: TypeScript will not be aware of the things your library offers - so you have to tell it. Do that with one of the following two options:

- Add the following entry to `src/typings.d.ts`: `declare module 'name-of-library'` (create that file if it doesn't exist)

- Add the following entry in **any** file where you use something from the library: `declare var whatIUse: any;`. Make sure to add it right below the import statements, before you create your classes or execute the other code.

Here's an example for the latter approach:

```
import 'lodash';
declare var _: any;

@Component({...})
export class MyComponent {...}
```

Integrating 3rd Party CSS Libraries

Integrating third-party CSS libraries like Bootstrap is also simple. You can either install the libraries via `npm install` or download the files (or use a CDN).

The libraries can then be included via one of the following three ways:

- Add a `<link href="path/to/file.css">` entry to your `index.html` file
- Add the path (relative from the `src/` folder) to `styles[]` in your your `.angular-cli.json` file
- Add an import to a TypeScript file (e.g. `main.ts`): `import 'path/to/file.css'`

The last option certainly looks strange but it works! You can import `.css` files into TypeScript files. They'll not really get imported but Webpack, which bundles everything behind the scenes, will take the file and ensure that it gets loaded in the `index.html` file in the end.

Important: For option two (add path to `styles[]`), you must use a path

relative from `src/`, **not** from the `.angular-cli.json` file. Here's the example for Bootstrap:

```
"styles": [
  "../node_modules/bootstrap/dist/css/bootstrap.min.css"
]
```

Share your Questions!

Do you have more questions? Please share them - either by leaving a feedback for this article or by sending a mail to feedback@academind.com. Make sure to subscribe to our newsletter to be informed when new interesting content is available!