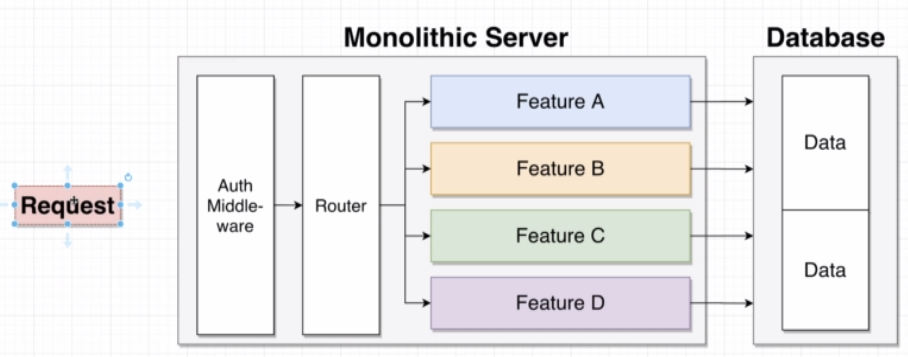
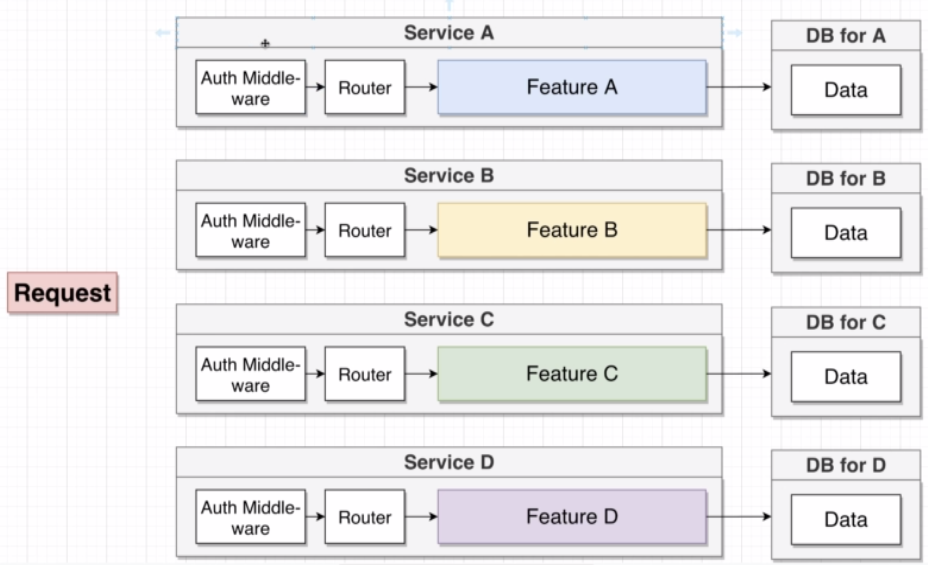
# Microservices with NodeJS and React

## Fundamental Ideas Around Microservices

### What is a MicroService?

Traditionally we have what is called a monolithic server shown below. It contains all logic to implement all features of our app.

Microservices have the idea of containing the logic for one feature of an application separately as shown below. All services are completely self-contained. Any failing of one service will not impact the other services.



### Data in Microservices

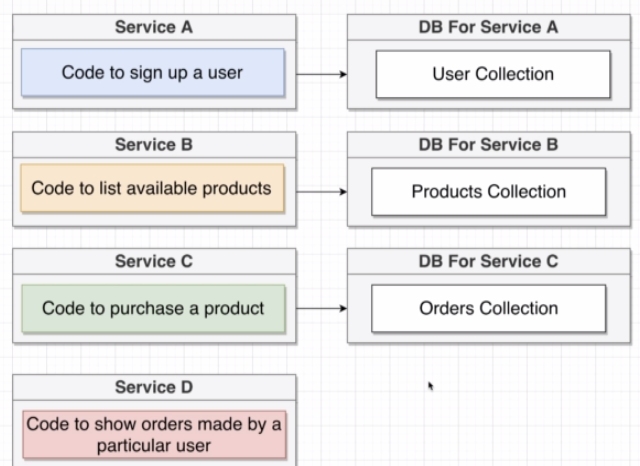
The biggest challenge of microservices is the data management between services, how we communicate data between the services.

As can be seen in the image above, each service has its own database, if it needs one. Services will never, ever reach into another services database. The reasons for this is the following:

* We want each service to run independently of other services.
* Database schema/structure might change unexpectedly.
* Some services might function more efficiently with different types of DB’s (sql vs nosql)

### Big Problems with Data

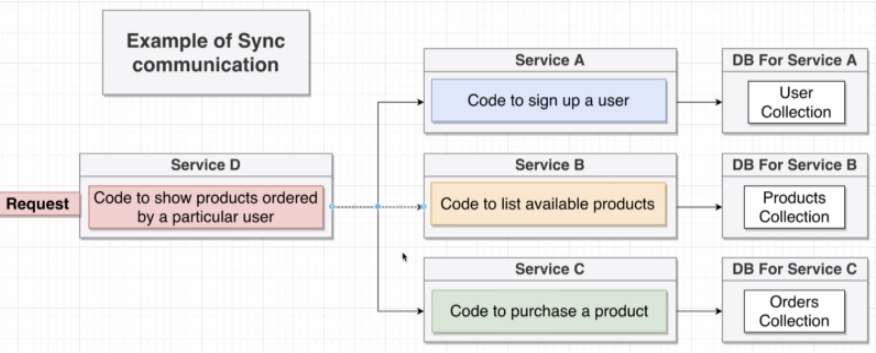
Imagine having a couple of services as shown below. They all have their own database with the data required for each service. Adding a new feature that requires data from all 3 databases becomes a problem since microservices don’t allow retrieval of data from another service’s database.



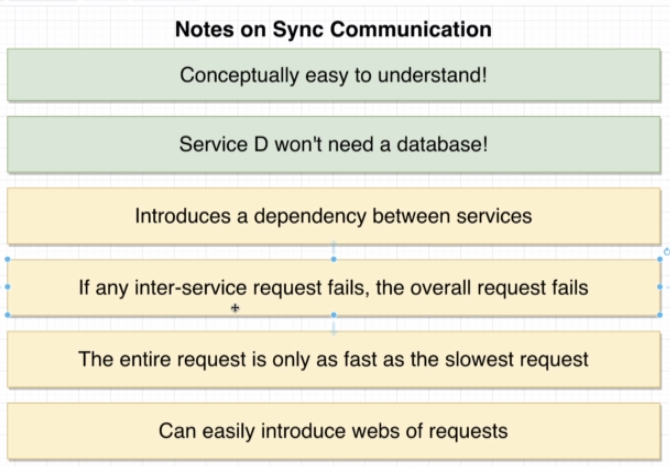
### Sync Communication Between Services

There are 2 general strategies that can be used to solve the data problem, Sync and Async, not to be confused with what these words mean in the JS world!

**Sync**: Services communicate with each other using direct requests. In the example above Service D would sent a request to service A, B and C to retrieve all data that is required to respond to a request to services D.



Below are the pros and cons for this strategy.



Even though the sync strategy has its upsides, it has some gigantic operational downsides.

### Event-Based Communication

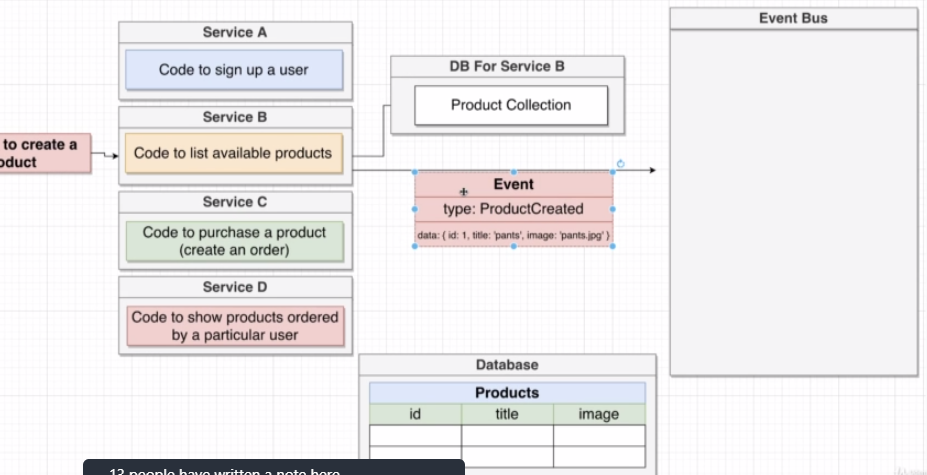
**Async:** Services communicate with each other using events.

There are basically **2 methods** that can be used to implement the async strategy.

**The first method** is the usage of what is called an Event Bus. Each service is going to connect to the event bus an emit or receive events passed onto it. Keep in mind that this creates a single point of failure!

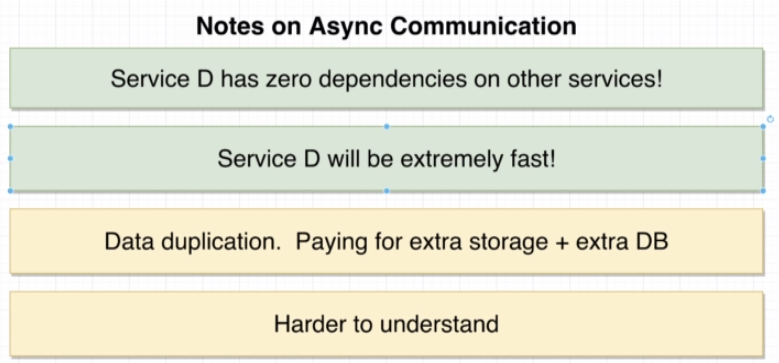
This style of async communication is not used much since it has all the downsides of the sync strategy.

**The second method** seems bizarre and inefficient. Basically we have to identify the data that is required to fulfill a response for a given request. In case of service D we need a products collection and a user collection containing all product id’s the user has ever ordered. To populate this database we, again, use an event bus.



Whenever a product is created, besides creating the product, an event is also created and put on the event bus. Service D will be watching out for this event and add the newly created product to its own database. This process also happens whenever a user signs up or a product is purchased. This will result in a fully populated database for service D. Whenever a request is passed to service D, its database will have all data to create a response.

The **pros and cons** are listed below:



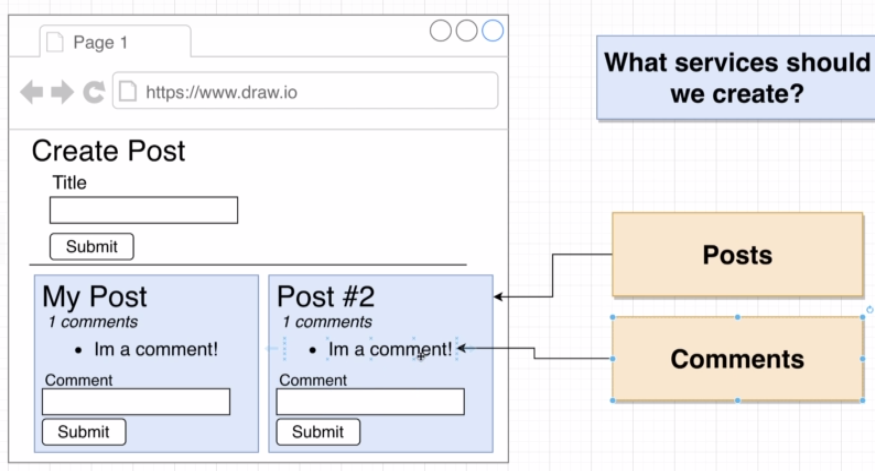
This course will focus entirely on this kind of strategy to handle microservices.

## Section 2: A Mini-Microservices App

### App Overview

This project will be built from scratch and is therefore not usable as a template for future projects. Reason for this is that there are a lot of packages that can be used to handle the microservices architecture.

The image below shows the setup for this application. A simple application will contain posts and comments for those posts. For the moment, a service will be created for each resource (post and comment).



### Project Setup

The project will contain 3 folders.

* The client being a React application using **npx create-react-app client**
* A Posts folder containing a package.json file and express, axios, nodemon and cors installed
* A Comments folder containing a package.json file and express, axios, nodemon and cors installed.

### Posts Service Creation

To get started, we create the express based posts service. The code below shows the 2 possible routes that can be utilized.

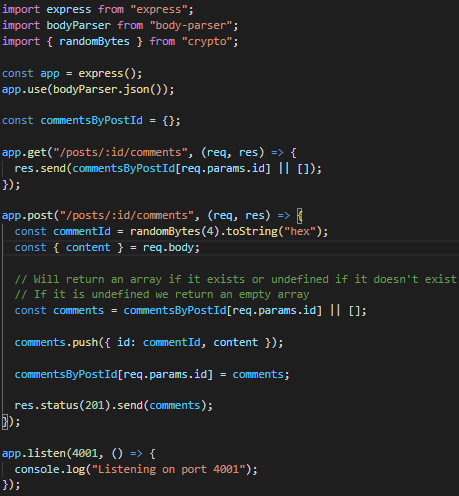
A couple of points to remember.

* We are storing our data in a plain object.
* Don’t forget to add a bodyparser to read json data.
* The randomBytes function is used to generate a random id, parsed to a hexadecimal value.

Using postman the functionality of the application is tested. Add a Content-Type (application/json) to the header to send to the post route.

### Implementing a Comments Service

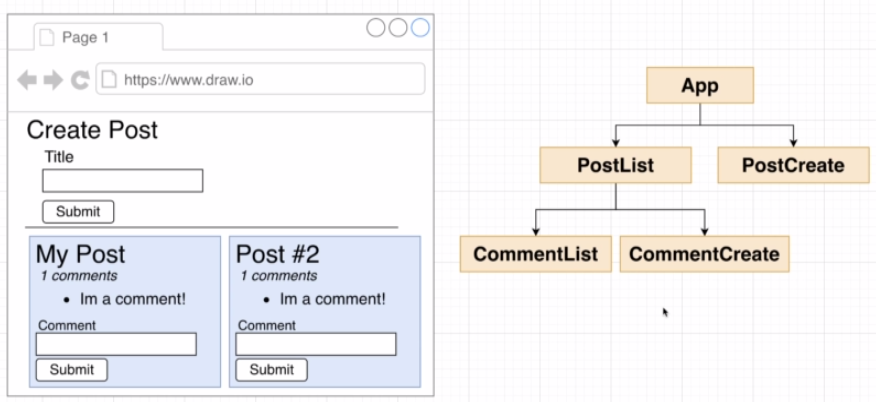
As with the posts service, an express based comments service is created, also containing 2 possible routes.

A couple of points to keep in mind:

* The commentsByPostId object will be populated, making it easy to retrieve all comments for a given post. A post id key will have an array with all the comments (or none).
* A non-existing post id key will return an empty array.

### React Project Setup

A basic React application is created. The axios package is added to facilitate http requests.



### Building Post Submission

The post creation component is shown below. This is a standard component using hooks to have a local state. An axios request will sent the post title to the posts service. Keep in mind that currently a CORS error will occur since no cors have been set on the backend posts service. See next lesson for the explanation.



### Handling CORS errors

Whenever we are trying to make a request to another protocol/domain/port, the cors policy within the browser will come into play. The browser will block any incoming response from another server unless the response has some very specific headers set that will have the browser allow this request.

What that means is that we have to do some additional configuration on the backend (posts and comments service).

The only thing required is adding the cors() middleware to the express server. This will set the correct headers and will fix the previous errors. These headers can be customized but for this purpose the default settings will suffice.

### Fetching and Rendering Posts

For fetching the posts, a new component is added, shown below.

The posts are fetched from the backend. We loop over the fetched object using the Object.values() method and map over that array to create jsx for each individual post.

### Creating Comments

As for the post creation, a similar code is used:

String interpolation is used to pass the postId, received as a prop from the PostList component, into the url.

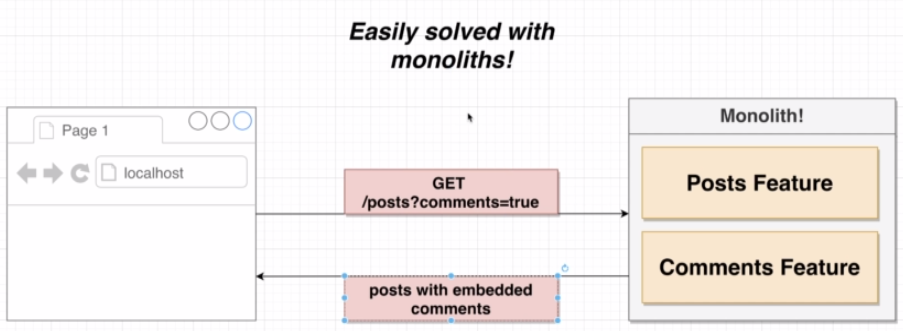
### Displaying Comments

As with the PostsList, we will fetch the comments from the comments service, and loop over them to create a list element containing the content of the comment.

The backend will return an array in this example.

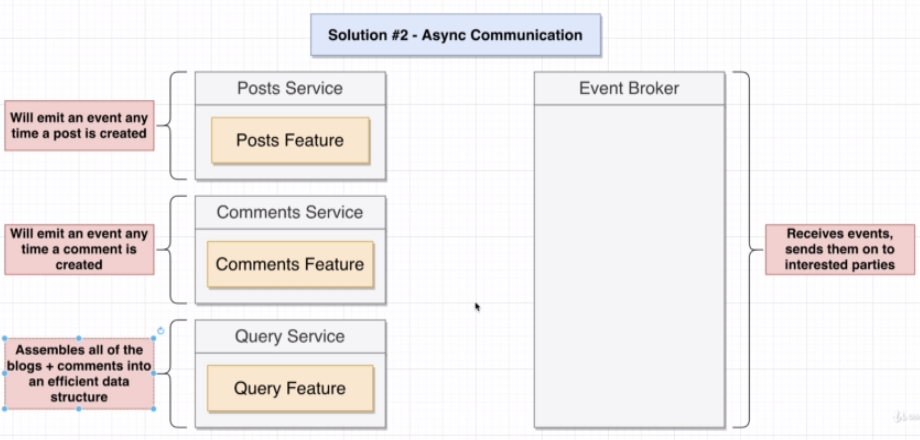
### Request Minimization Strategies

At the moment, we will send a request to our backend for every single post to fetch all comments for that post. This scenario is incredibly inefficient. In a monolith approach this would be very simple.



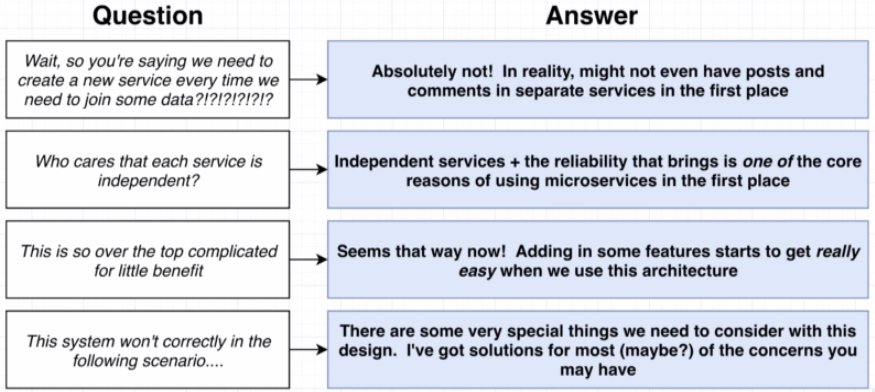
With microservices this isn’t possible so we need to figure out a way to efficiently fetch the data. **A first solution would be to use Sync communication**. The posts service will communicate with the comments service to fetch all comments for all posts. This however has the same pros and cons as discussed previously (pro: easy ; con: dependency between service, …).

**A second solution would be an Async solution.** To concept here is to create a third service that will handle events, dispatched by the posts and comments services, to create its own data structure that can then be called upon to fetch all posts and comments in 1 request.



The big advantage for the query service is having zero dependencies. If the posts or comments services fail, the query service will still function correctly. It will be extremely fast, since all that needs to be done is send back the data structure created whenever a new post or comment is created. This does however involve data duplication and is harder to understand.

### Common Questions Around Async Events

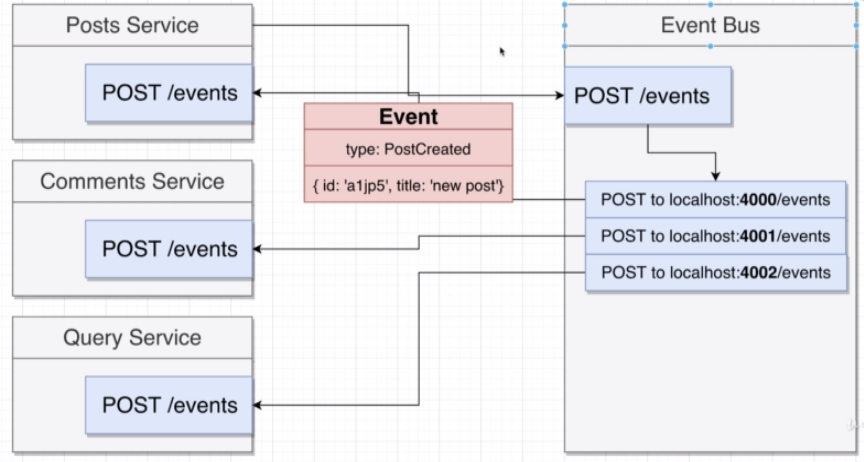


### Event Bus Overview

We are going to build a very rudimentary event bus for this small application from scratch. This is to give a feel for what an event bus does. In production, there are many open source implementations that can be used, and will be used later on in the course.

Our event bus will be build using express, basically a new service will be created to receive events from other services and publish them to listeners provided on the other listeners.

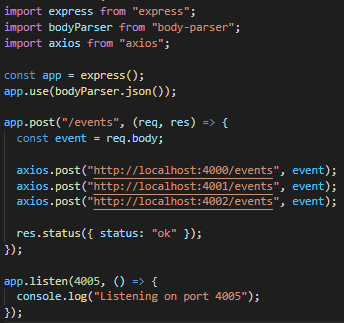
An example of the functionality is shown below



An event is passed to the event bus which will publish the event to all connected services, including the sending service. Keep in mind that this is a very basic implementation, just to show the basic principles of an event bus.

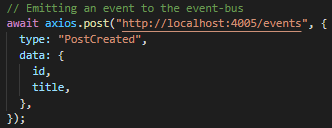
### A Basic Event Bus Implementation

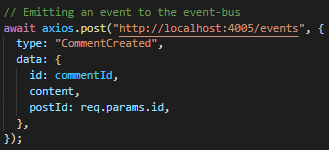
The implementation is shown below. This is a very basic express server, having an event route and publishing the same data to all other services.

One comment is that there is absolutely no error handling present.

### Emitting Events

For both the Posts and Comments service, an event is emitted whenever either resource is created. Below are both implementations:

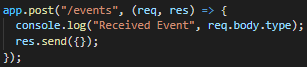




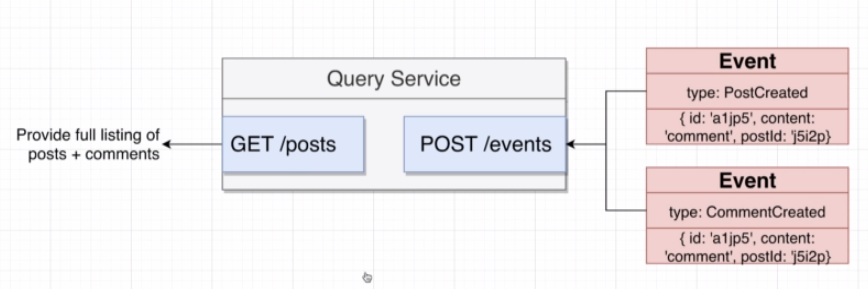
Currently no service is listening to these events, this will change when the query service is implemented.

### Receiving Events

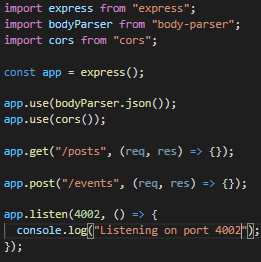
The event can now be received by both the posts and comments services. Below is the implementation for both services.



### Creating the Data Query Service

As discussed, this service will have a standalone database that will be populated with the data that is being sent over the event-bus into the query service. The below image shows both routes that will be needed to have a functional service.

As with the other services, the query service uses express to handle both routes.



### Parsing Incoming Events

The events are getting caught by the query service, which will use its data to populate its own database. The code below shows how this is done.



After these adjustements to the query service, we can directly get the posts and their comments in 1 request.

### Using the Query Service

Currently the application will still fetch the data for the posts from both the posts and comments services. Because our query service is completed we can now fetch all data in 1 request from this service.

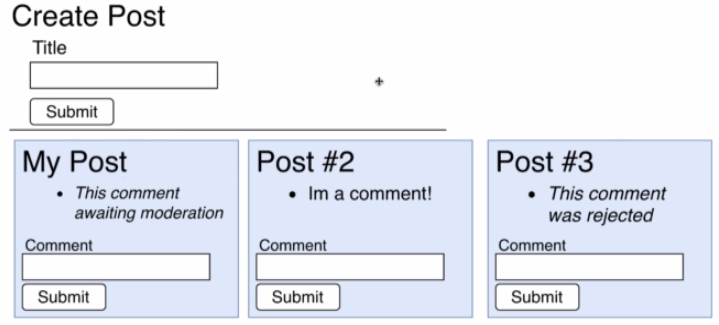
The application has been updated to facilitate these changes.

Shutting down both the posts and comments services will not affect the retrieval of the data required by the React application. Creating a post or comment is, however, impossible since those services are down but a great deal of the application is still running. This is the entire point of the microservices concept.

### Adding a Simple Feature

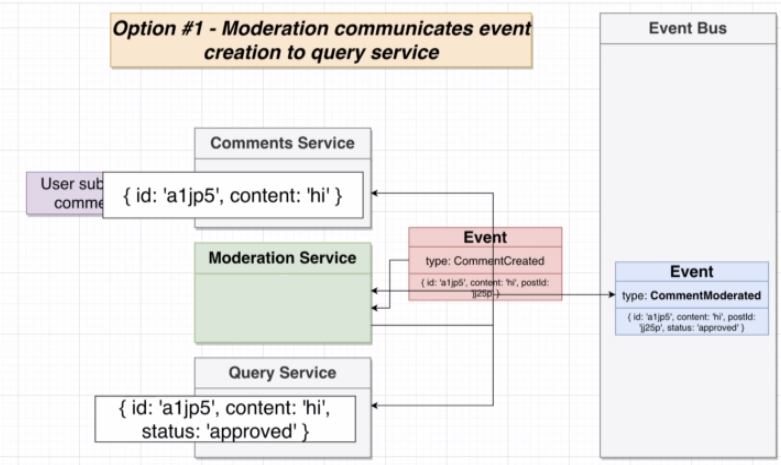
To showcase some complexities in a microservices application, a 4th service will be added, that has the responsibility of filtering a comment for the word ‘orange’. Obviously this can be implemented in multiple, easier ways, but for the purpose of example we will assume that this service will take a very long time. This service will be called the Moderation service.

Our database model on the query service needs to be changed to have a status added to the comment besides its id and content. This way React can display multiple states of a comment (approved, pending or rejected). The image below shows a possible clientside view.



### Issues with Comment Filtering

A first approach to handle this is shown in the image below.

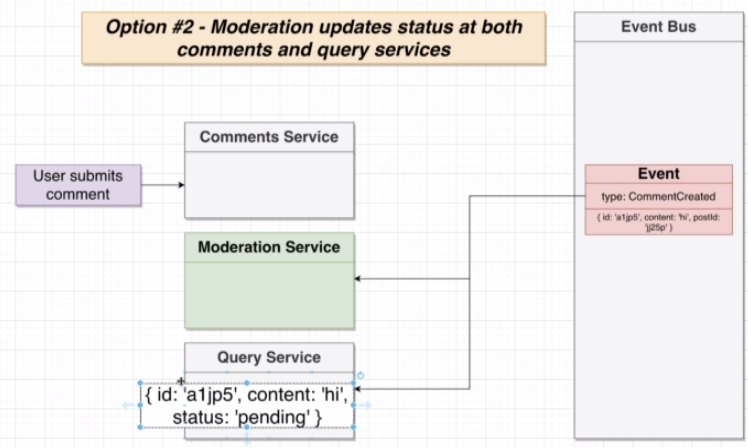


The moderation service will act as a middleware. Whenever a Comment is created, only this service will catch the event from the event bus and apply its logic (filtering) to it. Whenever this service has done its job, a new event will be send to the event-bus and passed to all other services. The query service will catch it and update its database.

The biggest drawback to this approach is the UX of the application. Imagine the moderation service being a human moderator. It can take a great deal of time before a comment is moderated, therefore the query service will not receive the newly created comment and a client can’t request the new comment. Another approach is required.

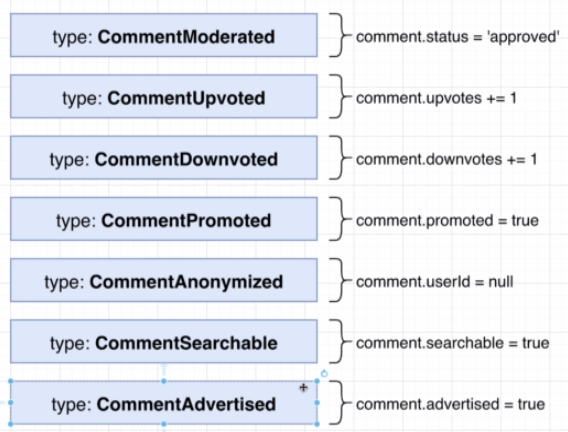
### A Second Approach

The second approach is a bit more logical. Whenever a new comment is created, both the moderation and query services processes the event. The query service will add the comment with a status of pending while the moderation service will apply its logic.



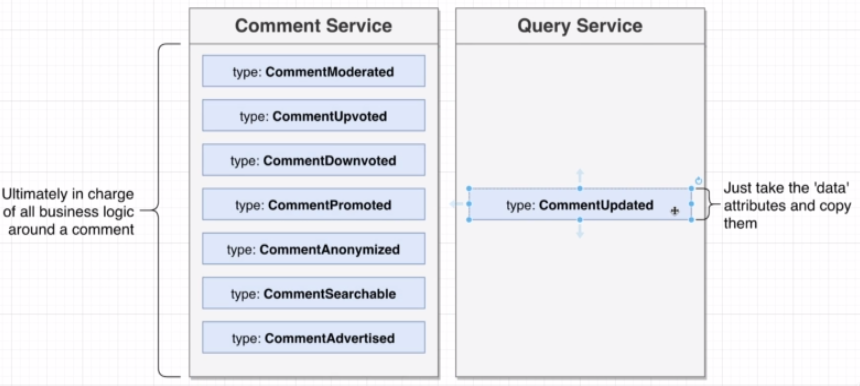
Whenever the moderation services has completed its logic, a new event is emitted to the event bus which will get caught by the query service, updating that comment to approved or rejected.

This might seem a good solution, however, we have to consider something. This application represents a very simple logic that will not drain any resources. Imagine having multiple events that get send to the query service, all of which need to update the database.

This creates a scenario where every possible way of changing a comment has to be handled by a ton of services (in case multiple services use comments). This is not what we want so we need a third approach that will handle resource updates better.

### How to Handle Resource Updates

The final solution will be to have the comment service handle all the business logic around a comment and send a generic event to the query service. The query service doesn’t have to interpret this event, it just has to update the database, regardless of what was changed.



Keep the following workflow in mind:

1. A comment is created which will emit the CommentCreated event to both the moderation and query service (status pending of course).
2. Whenever the moderation service has finished moderating the comment, it will emit the CommentModerated event to the comment service.
3. The comment service will emit a generic event, called CommentUpdated, to the query service which will simply update the entire comment in the database.

The next lessons will implement this behavior.

### Creating the Moderation Service

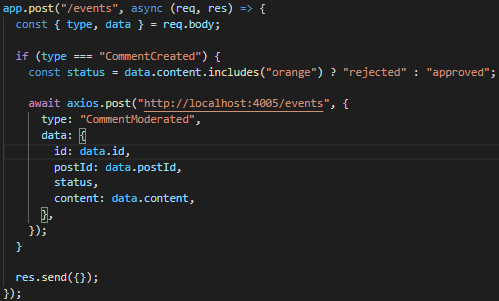
We start out by adding the moderation service containing 1 route path called ‘/eventes’. The next lesson will contain further logic.

### Adding Comment Moderation

Whenever a comment is created, a status needs to be added to the comment. Both the comment and query service have been updated to reflect the status of a comment.

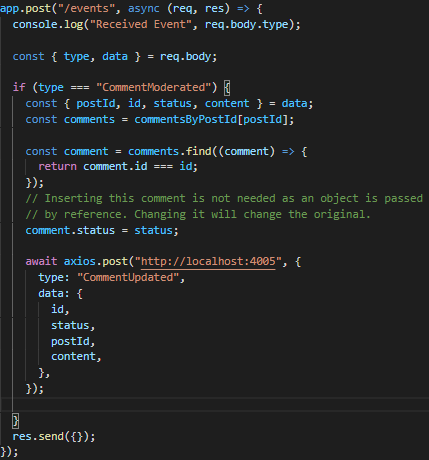
### Handling Moderation

The moderation service catches the CommentCreated event and applies its logic to it (checking if it contains the word ‘orange’). It then changes the status propertie to rejected or approved depending on the check being performed and emits an event back to the comment service. Below shows the code for the moderation logic. The updated comment is emitted to the event bus.



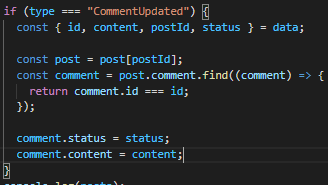
### Updating Comment Content

The comment service will catch the previously emitted ‘CommentCreated’ event to change its local comment. The code below shows this behavior. After it has been changed, the comment service will emit a generic ‘CommentUpdated’ event that the query service will catch.



### A Quick Test

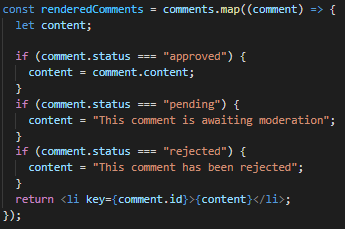
Before testing, the CommentUpdated event is caught by the query service to alter the local comment.

.

After testing, the code appears to be working, next the frontend needs to be updated to handle the status of a comment.

### Rendering Comments by Status

The frontend has been updated to use the status of each comment for rendering.



If we emulate a longer moderation response by shutting down the moderation service, we can see that this will effectively show the comment as “awaiting moderation”.

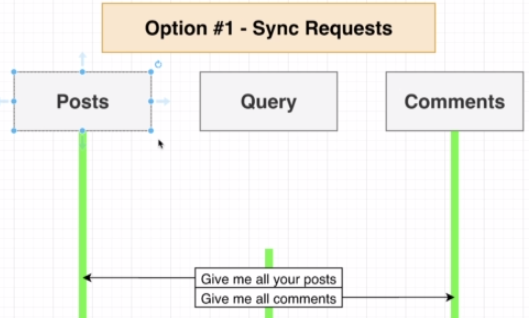
A problem that this creates is that after restarting the service, that event will be lost and the query service will not receive a moderated comment. It will remain “awaiting moderation”.

### Dealing with Missing Events

There are a couple of ways that we can resolve a service being down for a given period, resulting in that database not being up to date.

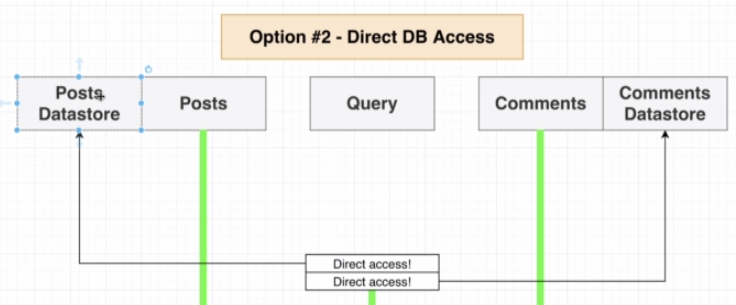
**Option #1 – Sync Requests**

Whenever a service comes online, ask the other services to provide all data needed. The downside to this method is that both the posts and comments need to have an endpoint added to support this functionality.



**Option #2 – Direct DB Access**

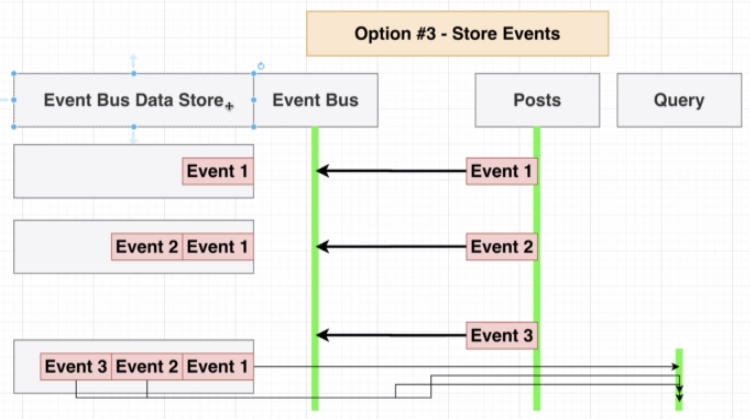
Provide direct access to the databases of both posts and comments services. Downside is that we need to add code to the query service potentially having to access different database types.



**Option #3 – Store Events**

If we had access to all of the events before the query service came online, basically the service could built up its local database to be synced with the other services.

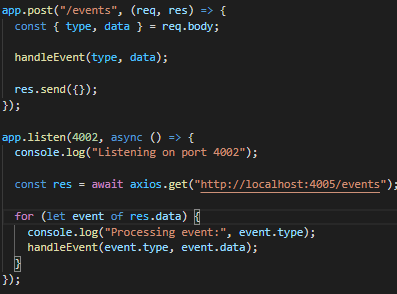
That is the solution to our problem. Whenever a service sends out an event, the event-bus service will store all the events it gets. This is shown in the image below.



This also solves the issue for a service going down for a short period of time. The service could ask the event-bus for all events that occurred during its time offline.

### Implementing Event Sync

The events will be stored in an array (normally this happens in a database). The code below shows how the events are processed.

The event handling is moved to a separate function. Whenever we start the service, a request will be send to the event bus asking for all events that occurred.

The event-bus has also been updated with an array that gets filled with every event and a ‘get’ events endpoints that sends this array.

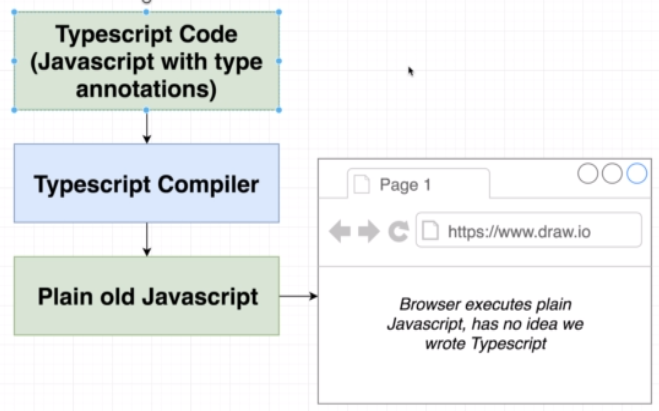
This concludes the logic part of the mini application. Next we will look at Docker which will be used for deployment.

## Section 3: Running Services with Docker

## Appendix A – Basics of Typescript

### TypeScript Overview

Typescript is Javascript + a type system. The TS type system will help us catch errors during development. It uses ‘type annotations’ to analyze our code. The type system is only active during development and does not provide any performance optimization.



Typescript gets compiled to plain javascript. It is only used as a development tool and the browser/node will only see plain javascript. At its core it’s a helper that helps us catch errors.

### Environment Setup

In order to use typescript we have to install the typescript package and ts-node ( a cmd line tool). The typescript package enables us to use the tsc (typescript compiler) command.

### A First App

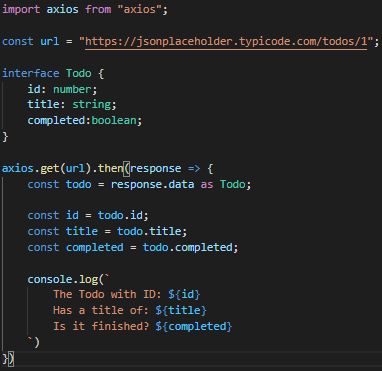
A project has been created to fetch data from a json api using axios.

### Executing TypeScript Code

A typescript file uses the .ts extension. As noted earlier, we cannot execute a typescript file directly. First it needs to be compiled to javascript.

Compiling a typescript file to javascript can be done with the following command “**tsc index.ts**”, which will create the index.js file that can be executed using “**node index.js**”. Because the ts-node package was installed, this can be done in 1 command. “**ts-node index.tx**” will compile and execute the file.

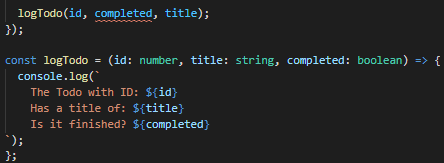
### Catching Errors with Typescript

During development Typescript can be used to catch errors. The code below shows an example of this.

An interface can be used to define a structure that a certain piece of data has. The response from the API will have a userId, id, title and completed property. By defining that within the Todo interface and explicitly setting the response.data as that interface, an error will be given whenever we try to access a property on todo other than id, title or completed.

### Catching More Errors

Typescript can also be used to define a type to the arguments passed to a function. The example below shows how this works.

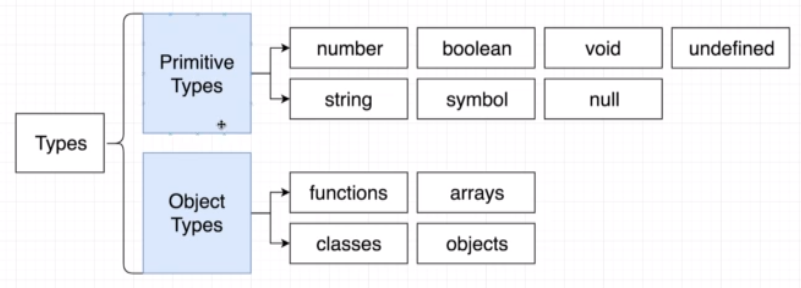
Because the title has to be of type string, an error will be shown when passing, in this case, a boolean value to the title parameter.

### Types

A type in typescript is an easy way to refer to the different properties + functions that a value has. Every value in typescript has a type. Keep in mind that we can create a custom type by creating an interface that defines a value.

### More on Types

There are 2 kind of types:

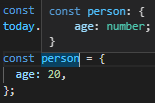


The object types can be used by a developer to define a custom type. We use types to analyze our code for errors and allow other developers to understand what values are flowing around our codebase.

### Examples of Types

A couple of examples are shown below:

The today variable is of type Date. Typescript can use this to show a list of its available methods and properties.

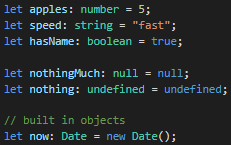
An object is shown as such. It does not have an interface connected yet, so it shows all properties.

Types are going to be used everywhere. This will enhance the troubleshooting drastically.

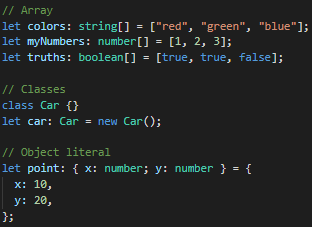
### Type Annotations & Type Inference

Type annotations refers to us telling typescript the type of a value. Type inference refers to typescript guessing what type a value is.

### Annotations With Variables

To the left are a couple of examples regarding type annotations. Whenever we try to assign a string to a value that has a number annotation, an error will be shown.

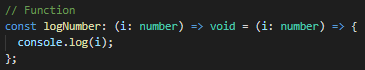
### Object Literal Annotation

To the left are examples of object literal annotations. Keep in mind the ; between different properties of an object annotation.

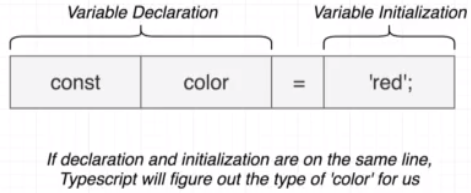
The rest is self-explanatory.

### Annotations around Functions

Functions are a little bit tricky. We need to provide the correct annotations for the arguments a function is passed as well as an annotation for its return type (void of nothing is returned).



### Understanding Inference

Type inference refers to typescript figuring out what type a variable is based on the value it is assigned.

We will use type inference as much as possible. We will only use type annotations:

* When we declare a variable on one line then initialize it later
* When we want a variable to have a type that can’t be inferred
* When a function returns the ‘any’ type and we need to clarify the value

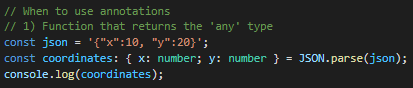
### The Any Type

To explain the any type, we can use the JSON.parse() method. The type that is being returned by that method is depending on the string that we pass to it. Therefore, typescript cannot infer what type it will return. For this reason typescript will infer the ‘any’ type.

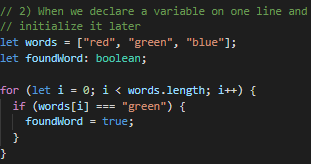
Keep in mind that:

* “any” is a type, like ‘string’ or ‘boolean’
* TS has no idea what this is and can’t check for correct property references
* **This type must be avoided at all costs**

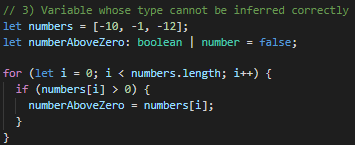
### Fixing the ‘Any’ Type

We fix this problem by adding a type annotation.

### Delayed Initialization



### When Inference doesn’t work

Whenever multiple types can be applied to a variable (bad practice), we can use type annotations to define those.

### Annotations Around Functions

Annotations or inference regarding functions is very similar to variables. The annotations tell Typescript what type of arguments a function will receive and what type of values it will return. Type inference for functions refers to Typescript to figure out what type of value a function will return, NOT infer the type of the arguments passed.

The following code shows an example of this:



The arguments are annotated directly followed by the type of the return value.

### Inference Around Functions

The only part of a function that can be inferred is the return value. If we look at the code above, Typescript will literally read the function and infer that the return value will be a number (if we don’t add the type annotation for the return value).

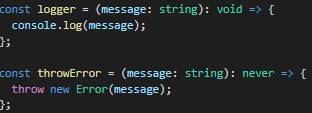
This however is bad practice, we will always add type annotation for the return value.

### Annotations for Anonymous Functions

Anonymous functions basically have the same annotation rules. Arguments followed by the return type.



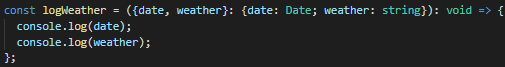
### Void and Never

A function that doesn’t return anything uses the void annotation.

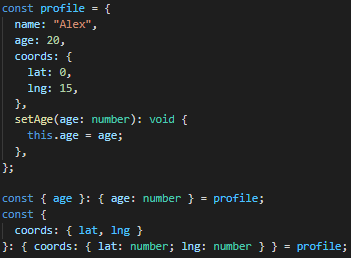
If a function doesn’t reach completion, never is used. This is the case for the throwError function.

### Destructuring with Annotations

Whenever we use destructuring, it’s important to first do the destructuring and then the type annotations, as shown below.



### Annotations Around Objects

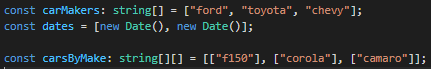
Adding annotations to objects works mostly the same as before. Important to note is that when destructuring properties from an object and add a type annotation to this property, always use the structure of the object we are destructuring from.

{age}: {age: number} is correct

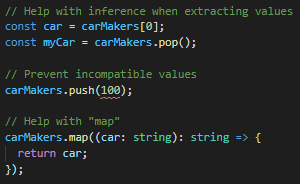
{age}: number is INCORRECT

### Arrays in Typescript

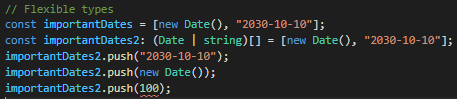
Typed arrays are arrays where each element is some consistent type of value.

The annotations are not required. They are inferred by Typescript.

### Why Typed Arrays

Typed arrays can help with the examples shown to the left. In regards to the map() function. Because Typescript knows the carMakers array is of type string, the map() function also knows it receives a string (car:string can be left out). This way the car variable will have the correct autocomplete (string methods).

### Multiple Types in Arrays



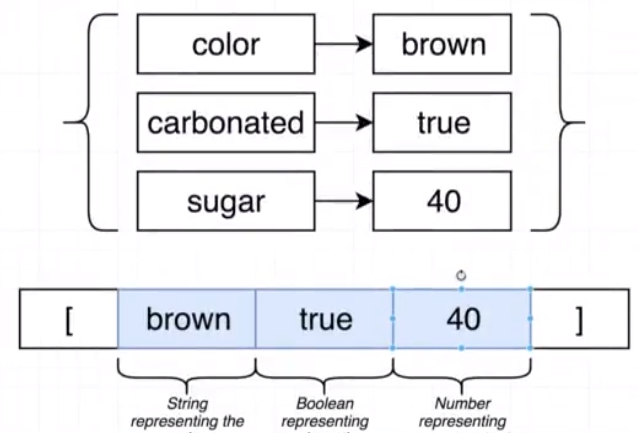
Using multiple values in an array can be done using the | symbol.

### When to Use Typed Arrays

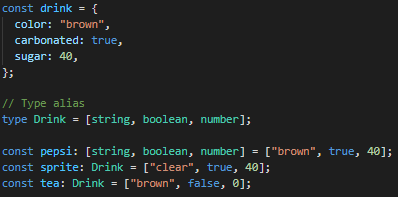
Any time we want to represent a collection of records with some arbitrary sort order.

### Tuples in Typescript

A tuple is an array-like structure where each element represents some property of a record.

As an example we can imagine an object representing a drink. We can turn this into a tuple by putting the values into an array, in a specific order! This is what a tuple is.

### Tuples in Action

The code to the left shows an example of a tuple.   
By adding an annotation in the form of [string, …] we are essentially turning the array into a tuple, indicating that the order is important.

A type alias can be used to replace this notation so that it doesn’t have to be typed out completely.

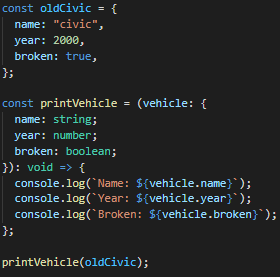
### Why Tuples?

Using an object is much clearer to work with than working with tuples.

### Interfaces

An interface creates a new type, describing the property names and value types of an object.

### Long Type Annotations

The example shows how we would implement type annotations for a function. It is clear that this quickly becomes very verbose. A solution for this is interfaces…see next lecture.

### Fixing Annotations with Interfaces

The solution to the annotations is using an interface.

Typescript will check that the ‘oldCivic’ variable has the properties defined within the interface.

### Syntax Around Interfaces

We can use any type inside of an interface, we are not limited to primitive values. We can also express functions within an interface.

### Functions in Interfaces

In the example above, the Vehicle interface could also only have the summary method. Even if the oldCivic object contains more properties, this does not matter. Typescript will only check if the object has the summary function included.

Only having the summary method within the interface, brings up the question to make this a more generic interface. The code below shows the refactoring of the interface so that it can be used on more than just a vehicle.

### Code Reuse with Interfaces

Any object containing the summary method returning a string can use the Reportable interface.

### General Plan with Interfaces

A general overview of the Reportable interface is shown below.



The general strategy for reusable code in typescript is:

* Create functions that accept arguments that are typed with interfaces
* Objects/classes can decide to ‘implement’ a given interface to work with a function

### Classes

A class is a blueprint to create an object with some fields (values) and methods (functions) to represent a ‘thing’.

### Basic Inheritance

A class can inherit properties and methods from a parent class by using the extends keyword.

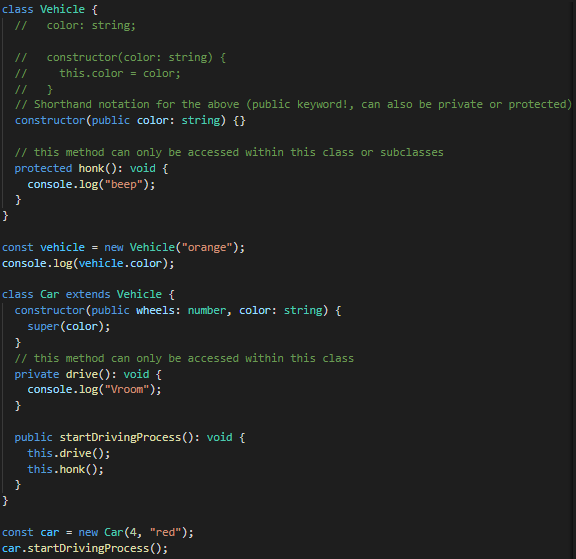
### Class Method Modifiers

Modifiers are keywords we can place on different properties or methods in classes (public, private, protected). This is a Typescript feature, not a Javascript feature.

* Public (default): this method can be called anywhere, any time.
* Private: this method can only be called by other methods in **this class**
* Protected: this method can be called by other methods in **this class**, or by other **methods in child classes**.

The reason modifiers are used is not to add a layer of security to our application. It is to restrict access to methods for other developers.

A full example is shown below



### Where to Use Classes

Classes are used to improve code reusability.

### App Overview

As an example project, we will create an application that will randomly generate a user and company and show those on a map.

For this project a command line tool called ‘parcel-bundler’ is going to be used (similar to ts-node) to get Typescript running in the browser.

### Parcel in Action

The globally installed parcel command line tool will receive an index.html file with a script tag pointing to a Typescript file. It will then compile to Javascript and start a server providing the webpage. The command is as follows:

**parcel index.html** (within the correct folder)

### Project Structure

The files that will be created are:

* User class file
* Company class file
* Map file

### Generating Random Data

For generating random data, the ‘faker’ package is used.

### Type Definition Files

Importing a 3rd party library requires an extra step for Typescript to work correctly. Essentially, typescript wants to know what kind of data is flowing around inside an application. For this a type definition is used.

A type definition file is going to tell the Typescript compiler all of the different functions that are available inside a Javascript library (package), what type of argument they take and what type of value those functions return.

Keep in mind that some packages, by default, have a type definition file included. This is the case for axios but not for faker. For this reason we have to include it manually.

Most type definition files have already been created by a random person. These files can be retrieved from the ‘Definitely Typed’ project. By installing the @types/faker package we retrieve the file.

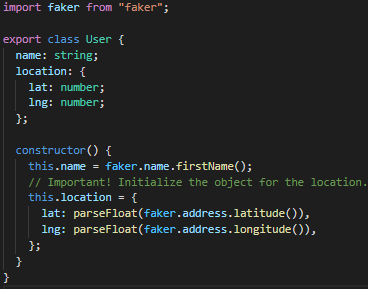
### Using Type Definition Files

Hovering over the faker package and pressing ctrl, enables us to open the type definition file. This can act as a guide to see all the available options.

### Export Statements in TypeScript

Convention in Typescript is not to use default exports. Use normal exports and import with { }.

Below we can see the User class and its usage in the index.ts file.



### Defining a Company

The Company class is added to the project, similar to the User class. See the project for its implementation.

### Adding Google Maps Support

To add support for google maps, the following steps need to be taken

* Add a project to the google developers platform
* Enable the maps for javascript API
* Create an API key
* Add a script tag to index.html with the generated API key

### Google Maps Integration with Typescript

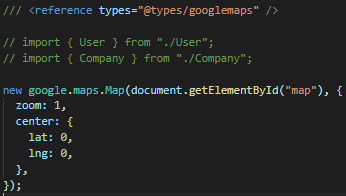
Using the Google Maps within our project requires an additional step. Adding the script tag to index.html, adds the google object to the global variable, however, we need to inform Typescript of this so that we can use it.

Using a type definition file will enable us to use the google object within the project.

**npm i @types/googlemaps**

### Exploring Type Definition Files

The code below shows how to display a map in the browser. Keep in mind that the div with an id of ‘map’ must be added to index.html.

The first line is to get the type definition working.

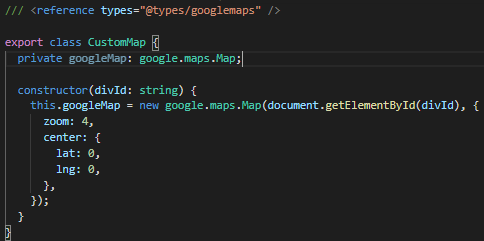
When we us ctrl + click type definition, we can see all available options the google object provides. It is essential to learn how to read these files in order to better understand how to use a given library.

### Hiding Functionality

Currently we are using the Map class that google has provided for us. While this is obviously the purpose, this enables any developer to call all available functionality of the Map class. In order to limit the access to this class, we can create a custom Map class that limits the functionality that can be used. The next chapter will handle this concept.

### Why Use Private Modifiers? Here’s why.

To facilitate the above functionality, we need to create a custom class that only allows certain methods to be publicly accessible. The code below shows this functionality.

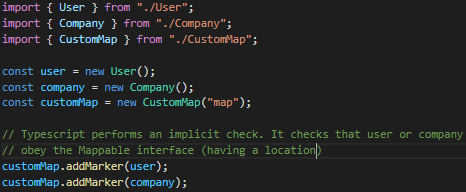
Setting the googleMap property to private prevents other developers from calling that object. Modifying this file itself obviously still results in another developer being able to access all methods the google object provides.

Another extra is that we can pass an id to connect the map to the element.

### Adding Markers

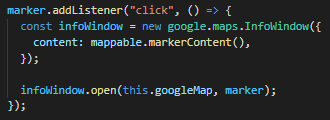
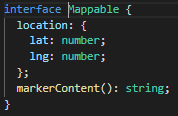
The code below shows the entire implementation for adding a marker. Important to note here is that an interface is used to check if the user or company have a location object with a lat/lng property that is a number. The reason this has been added is for extensibility. When, in the future, another class needs to add a marker to the map, all that that class needs to do is obey the interface that is required.





### Showing Popup Windows

A marker can be provided with a popup that shows additional details regarding the marker. In order for any class to obey the CustomMap it’s necessary to add the content to be shown in the popup to the interface. The implementation is shown below.



When creating a marker, we can add an event listener that will handle the popup showing (infoWindow). In order to make sure a class, that is passed to the addMarker function, has the markerContent() function, we need to add this to the interface.

### Optional Implements Clauses

The implements keyword is used to tell Typescript that a specific class has to obey the interface it implements.

export class User **implements** Mappable { }

This is used to get better error handling, so a developer is directly pointed to the class that is missing some properties or methods.