
Workshop
Intro to the MEAN Stack
Powered by BlackRock
MLH localhost
BLACKROCK®

1

Using your Web Browser,
Open this URL:

http://mlhlocal.host/lhd-resources

2

Click on the workshop you're attending, and find:

- Setup Instructions
- The Code Samples
- A demo project
- A Workshop FAQ
- These Workshop Slides
- More Learning Resources

MLH

MAJOR LEAGUE HACKING

Our Mission is to Empower Hackers.

65,000+
HACKERS

12,000+
PROJECTS CREATED

3,000+
SCHOOLS

We hope you learn something awesome today!

Find more resources: http://mlh.io/

What will you learn today?

1. The components of the MEAN stack: MongoDB, Express, Angular, NodeJs.
2. How Angular provides a framework for build web-pages.
3. How MongoDB stores data.
4. How Node and Express glue all the components together.

Table of Contents

In this workshop, you're going to gain an introduction to the MEAN Stack, which is a group of JavaScript tools you can use for full stack development.

Let's dive in!

What's Web Development?

Web developers design, build, and maintain websites.
Two main categories:

Front End, or Client-Side:

- Directly user-interactive website parts
- Technologies
  - HTML: Content
  - CSS: Layout
  - Javascript: Interaction

Back End, or Server-Side:

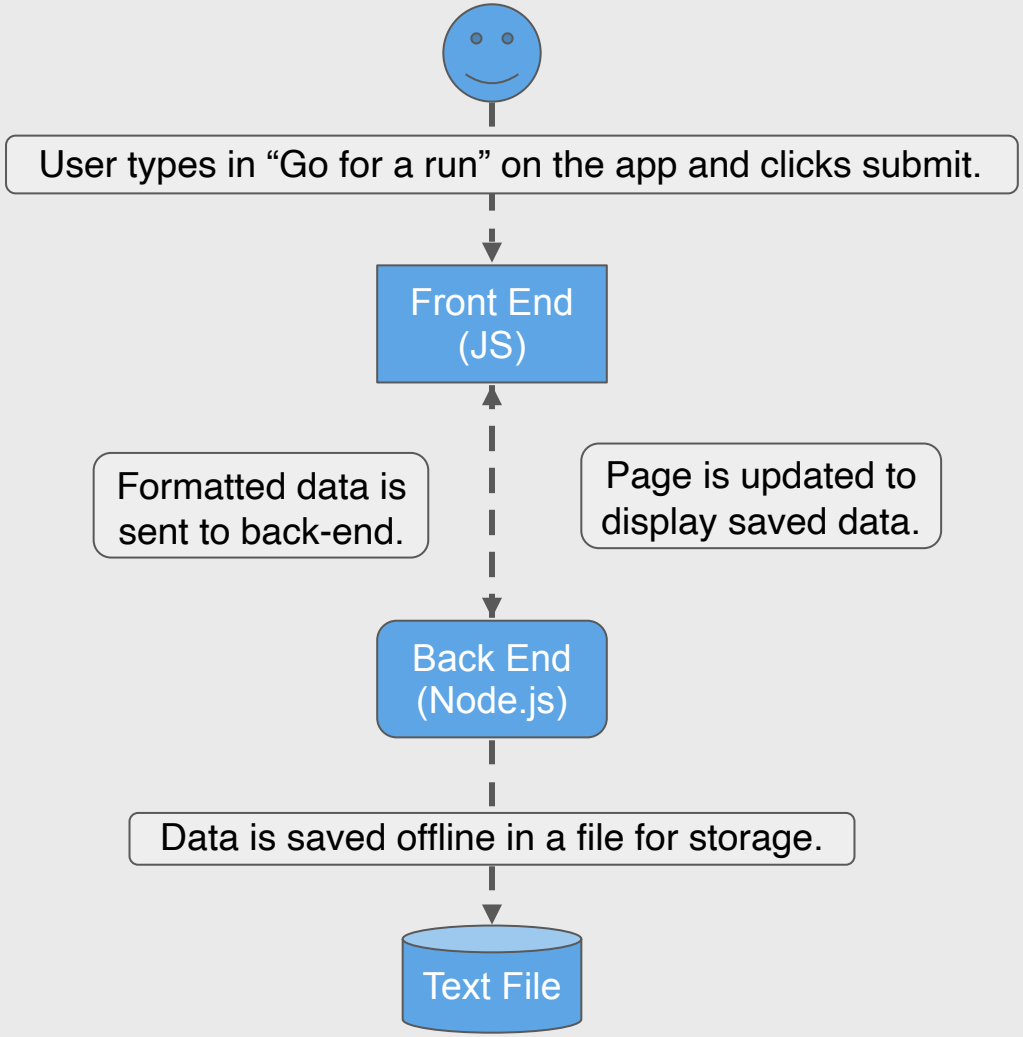
- User-hidden functionality
- Basically any language, but mostly Node.js, Ruby and Python

Key Term

Full Stack developer: Someone who is familiar with both kinds of development, and can work on a website from start to finish!

What's a Stack?

How does an app like a ToDo List work?



What's The MEAN Stack?

The MEAN Stack is a group of JavaScript libraries, frameworks, and tools that are usually used for large scale, production applications.

Acronymically

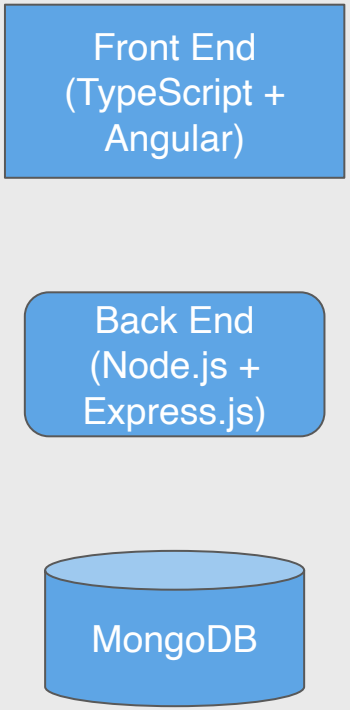
MongoDB: Store data offline in a fancy file

Express: Library to help communicate between Front End and Back End

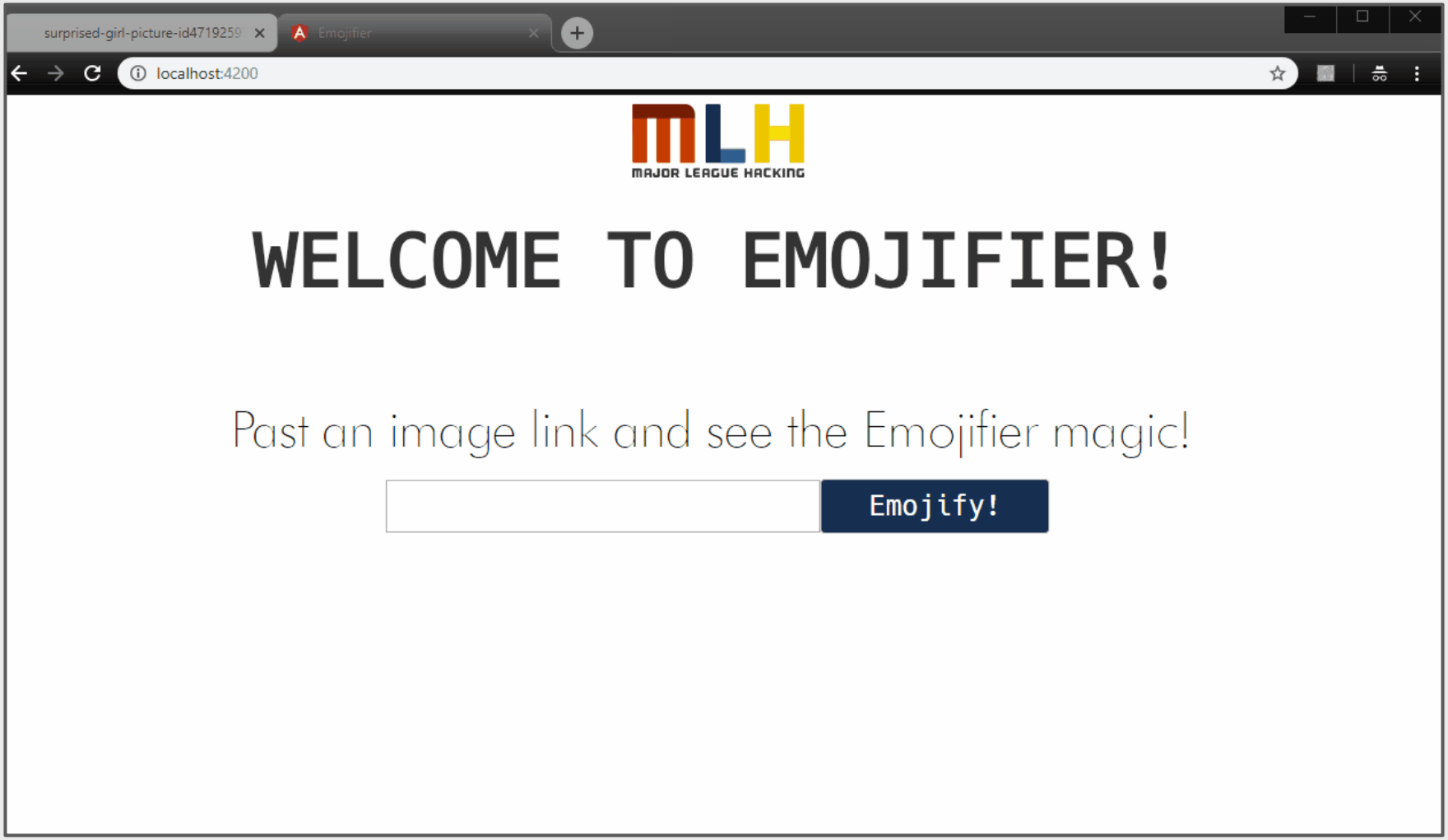
Angular: Framework to more easily make interactive webpages

Node.js: The glue between all the components. Serve the files and access the database

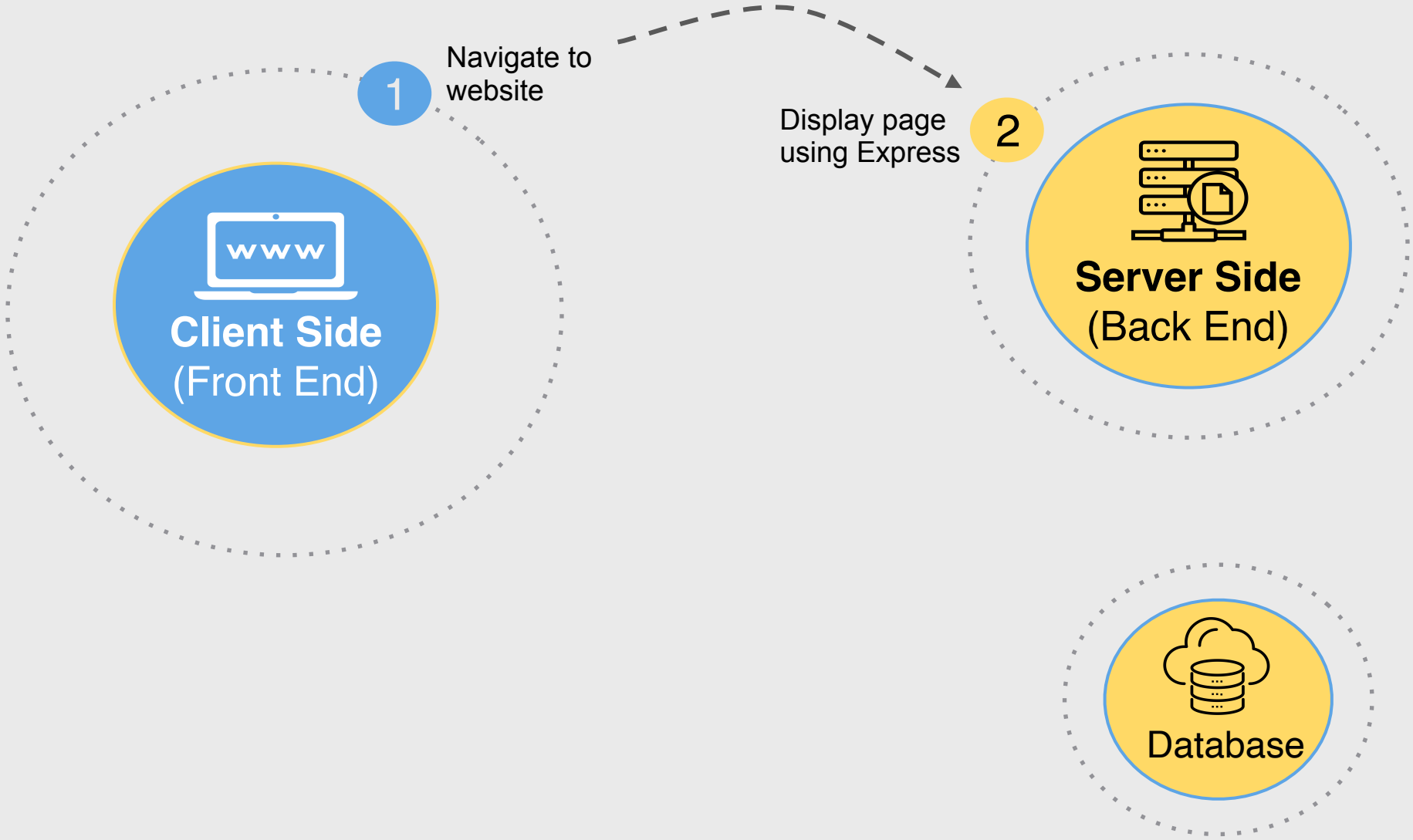
Practically



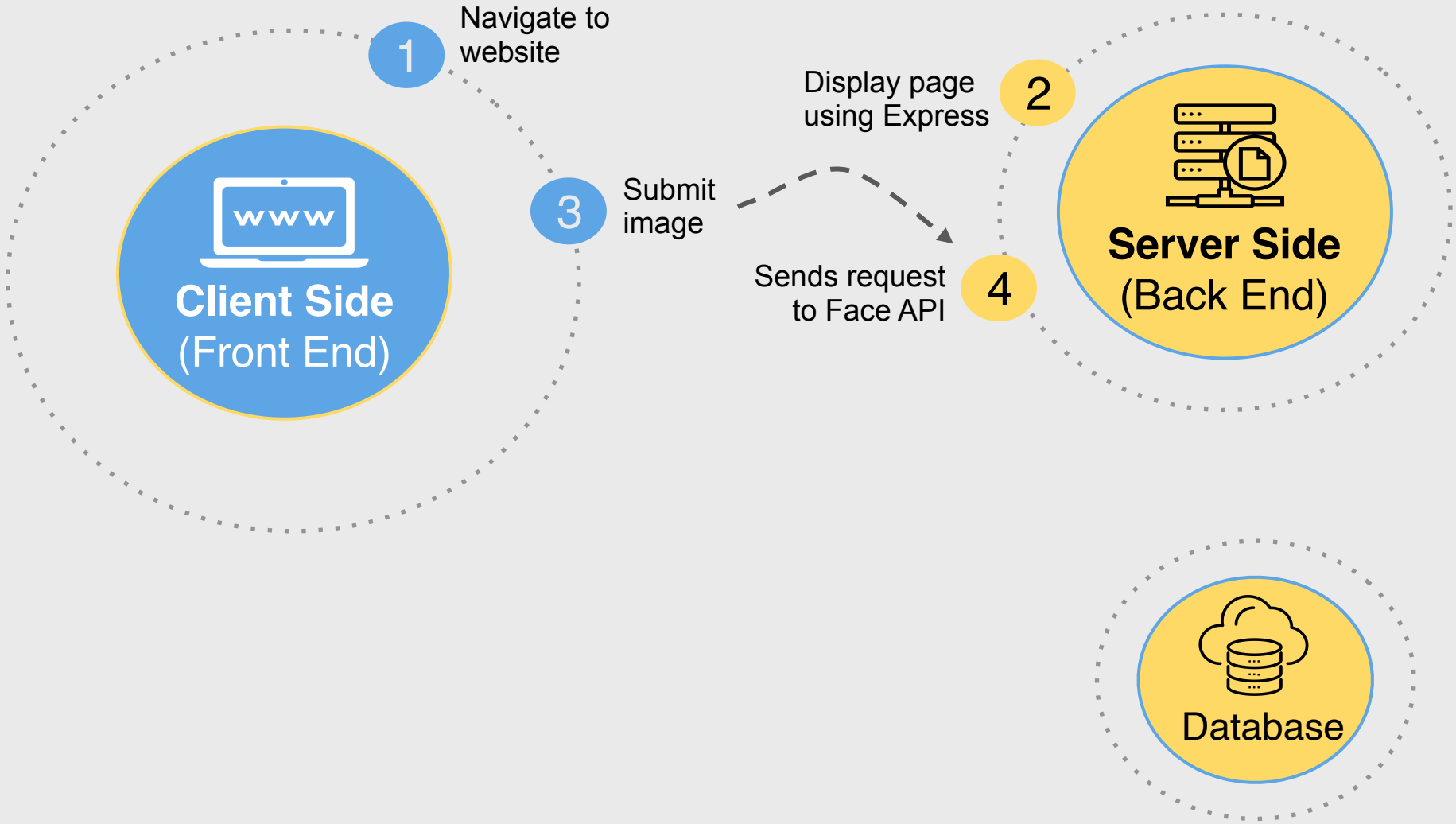
What You'll Be Building



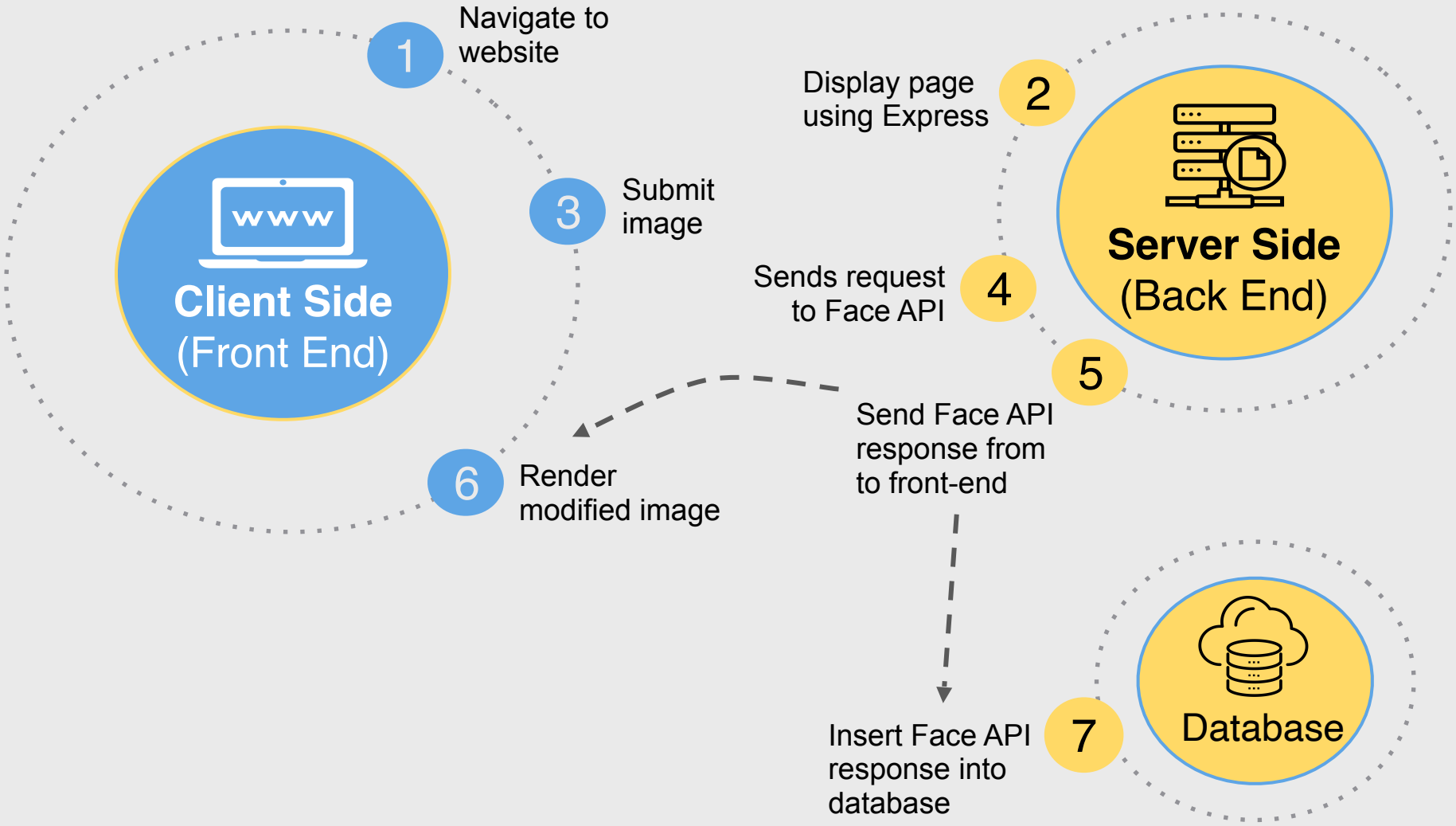
How It Works


1
Navigate to website
Display page using Express
2
www
Client Side
(Front End)
Server Side
(Back End)
Database

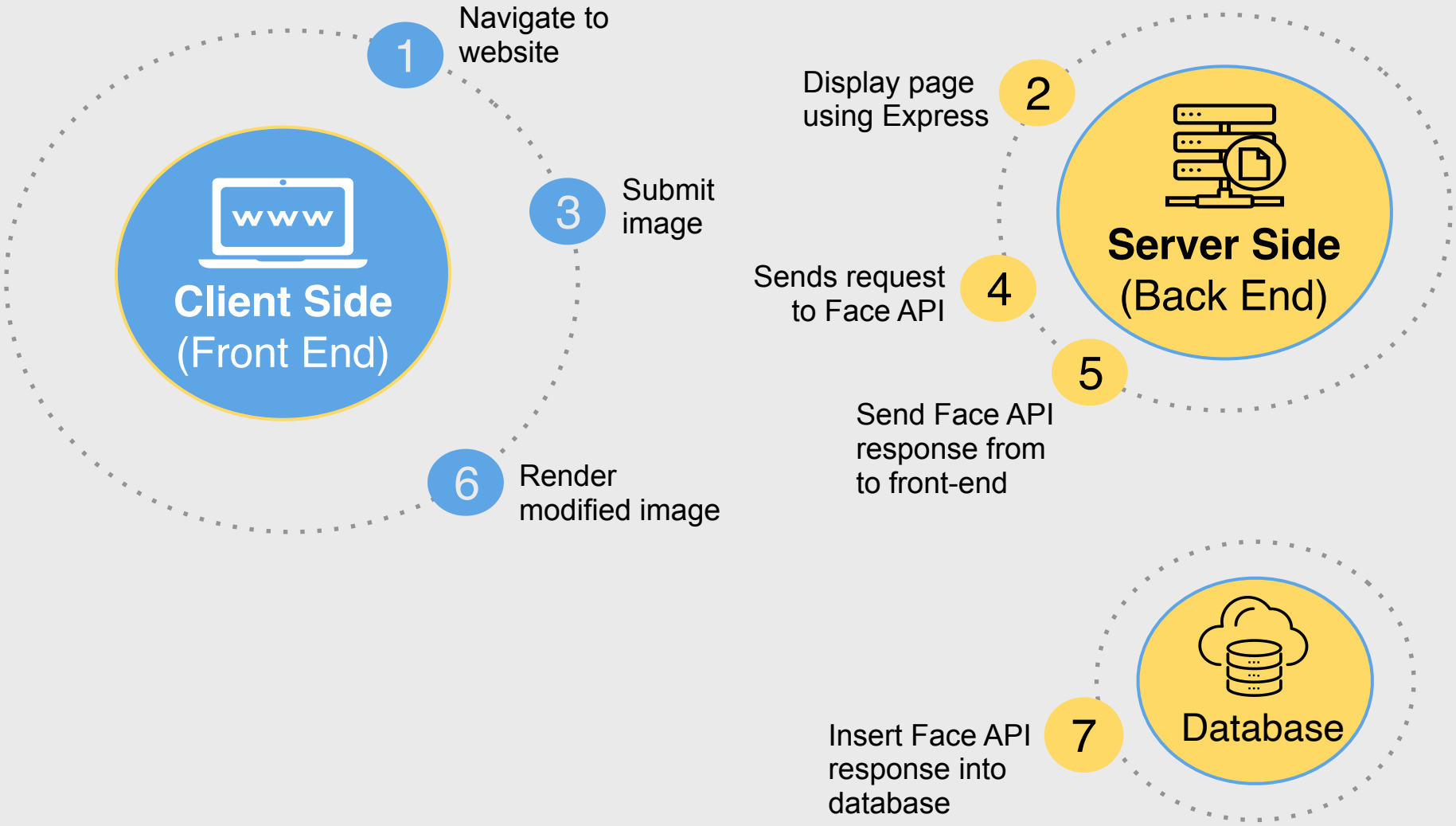
How It Works


1
Navigate to website
www
Client Side
(Front End)
3
Submit image
Sends request to Face API
4
Display page using Express
2
Server Side
(Back End)
Database

How It Works


1
Navigate to website
2
Display page using Express
3
Submit image
4
Sends request to Face API
5
Send Face API response from to front-end
6
Render modified image
7
Insert Face API response into database
www
Client Side
(Front End)
Server Side
(Back End)
Database

How It Works


1
Navigate to website
2
Display page using Express
3
Submit image
4
Sends request to Face API
5
Send Face API response from to front-end
6
Render modified image
7
Insert Face API response into database
www
Client Side
(Front End)
Server Side
(Back End)
Database

How You're Going to Build It

We'll use various tools to put this app together!

1. Install Node.js and MongoDB on your computer.
2. Download the sample code for the project.
3. Get developer credentials for the Microsoft Face API.
4. Explore the codebase for the project.
5. Write code for each part of the stack.
6. Test your app!

Now that you understand how the app works, let's get started!

We're going to spend about 20 minutes getting our computers ready for full stack JavaScript development.

Table of Contents

Install Node

The first thing you want to do is see if you already have Node installed.
Let's do that now!

1. Enter the command below in your Terminal. If you see output similar to that below, you have Node installed and you can skip to the section about installing MongoDB. Any version of Node starting with 10.0 will work.

Install Node

2. If you do NOT have Node installed, navigate to the URL that suits your computer and follow the typical installation process for your system.

http://mlhlocal.host/node-mac	Download for Mac
http://mlhlocal.host/node-pc	Download for PC
http://mlhlocal.host/node-linux	Download Page to Select Linux Version

Install MongoDB (Mac)

The easiest way to install MongoDB on a Mac is using Homebrew. First, let's check to see if you have Homebrew installed. **Note:** If you run into any errors during this section, make sure you have Xcode installed.

1. Enter the command below in your Terminal. If you see output similar to that below, you have Homebrew installed and can skip to installing MongoDB.

```
@localhost > brew --version
Homebrew 2.0.2
Homebrew/homebrew-core (git revision a506; last commit 2019-03-04)
Homebrew/homebrew-cask (git revision 4acd; last commit 2019-03-03)
@localhost >
```

Install MongoDB (Mac)

2. If you don't have Homebrew installed, navigate to the URL below and following the instructions on the website.

http://mlhlocal.host/homebrew

Install MongoDB (Mac)

3. Once you have Homebrew installed, enter the following commands to install MongoDB. You can also copy these commands from **`mlhlocal.host/mongo-mac`**.
4. Keep this terminal running in the background. Open a new terminal for the rest of the tutorial.
5. See the next slide if you run into errors while installing MongoDB.

```
@localhost > brew tap mongodb/brew

@localhost > brew install mongodb-community@4.0

@localhost > brew services start mongodb-community@4.0
```

Install MongoDB (Mac)

Troubleshooting

Note: Homebrew is very smart. If it gives you errors and suggests commands, use those commands! See an example below.

```
@localhost > sudo chown -R $(whoami) /usr/local/Homebrew
@localhost > brew tap mongodb/brew
Updating Homebrew...
==> Auto-updated Homebrew!
```

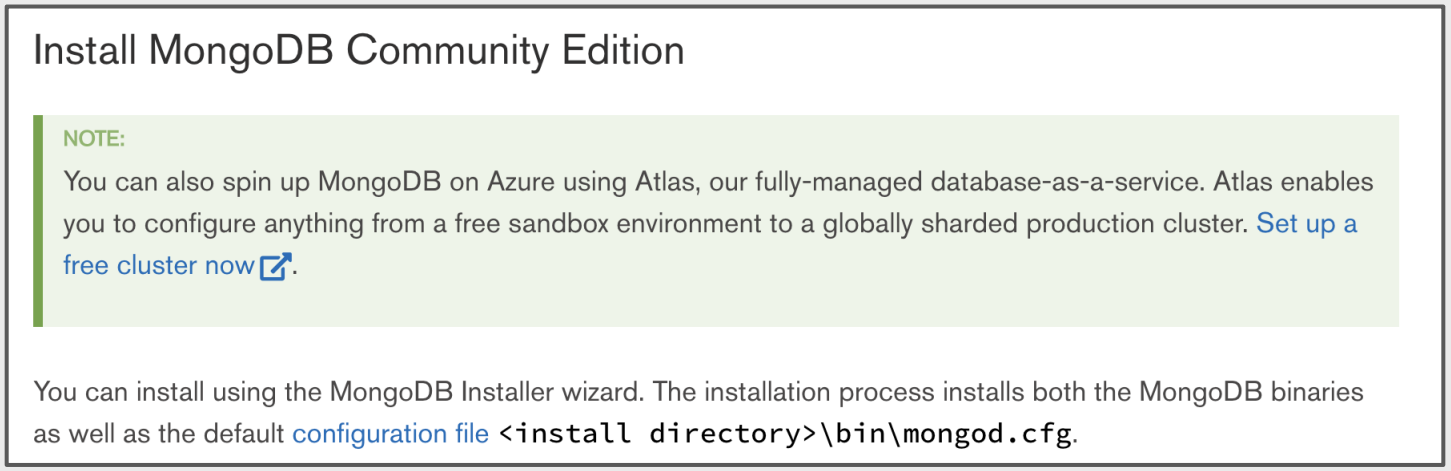
On my computer, I had to change the command above a little bit. If you get stuck here, ask others around you for help!

```
@localhost > brew tap mongodb/brew
Error: Can't create update lock in /usr/local/var/homebrew/locks!
Fix permissions by running:
  sudo chown -R $(whoami) /usr/local/var/homebrew
@localhost >
```

Install MongoDB (Windows)

1. Navigate to the URL below for instructions to install MongoDB.
2. Complete sections 1 through 3 on the page below.
    a. If you choose to install Mongodb NOT as a Service, you will have to restart your computer. However, both methods work for this workshop
3. Then, follow the instructions on the next slide.

http://mlhlocal.host/mongo-pc



Install MongoDB Community Edition

NOTE:
You can also spin up MongoDB on Azure using Atlas, our fully-managed database-as-a-service. Atlas enables you to configure anything from a free sandbox environment to a globally sharded production cluster. Set up a free cluster now.

You can install using the MongoDB Installer wizard. The installation process installs both the MongoDB binaries as well as the default configuration file `<install directory>\bin\mongod.cfg`.

Install MongoDB (Windows)

4. Open the command prompt (not PowerShell).
5. Create a directory and run MongoDB in it by running the commands below.
6. Keep this command prompt running in the background. Open a different terminal for the rest of this workshop.

```
$ cd C:\

$ md data\mongodb

$ "C:\Program Files\MongoDB\Server\4.0\bin\mongod.exe" --dbpath="C:
\data\mongodb"
```

Okay! Now that you have your environment set up, let's download the code for the project!

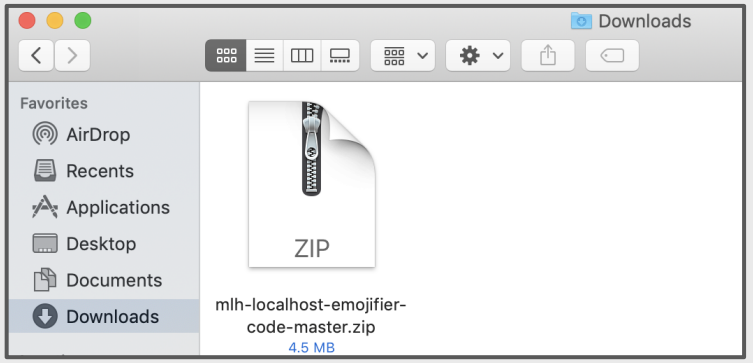
Download the Emojifier Starter Code

http://mlhlocal.host/mean-code

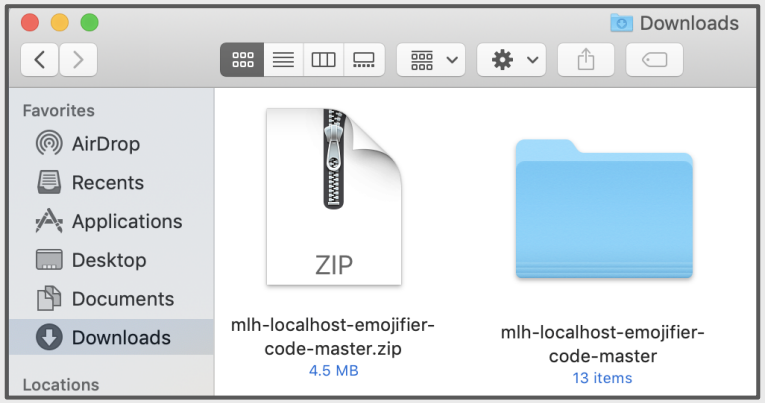
1. Navigate to the URL above.
2. A zip file of the code will be downloaded.
3. Follow the instructions on the next slide if you are on a Mac, or the following slide if you are on a PC.

Open the Emojifier Starter Code (Mac)

1. Starter code in a zip file was downloaded. Double click the download to unzip the file.

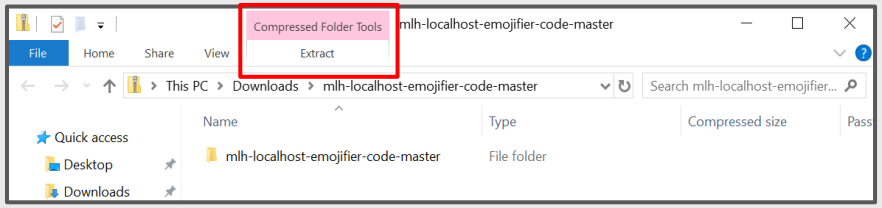


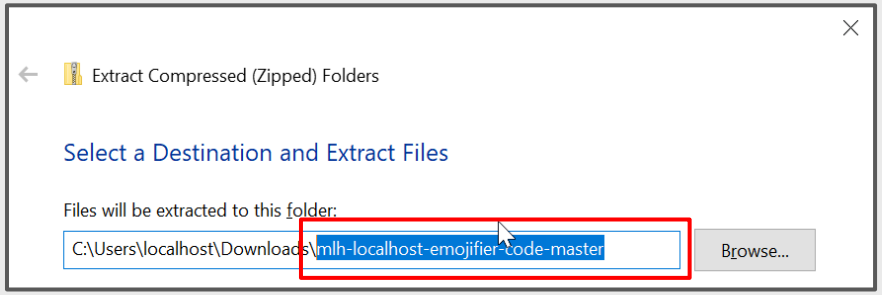
1. Enter the following commands in your terminal to navigate to the project folder.

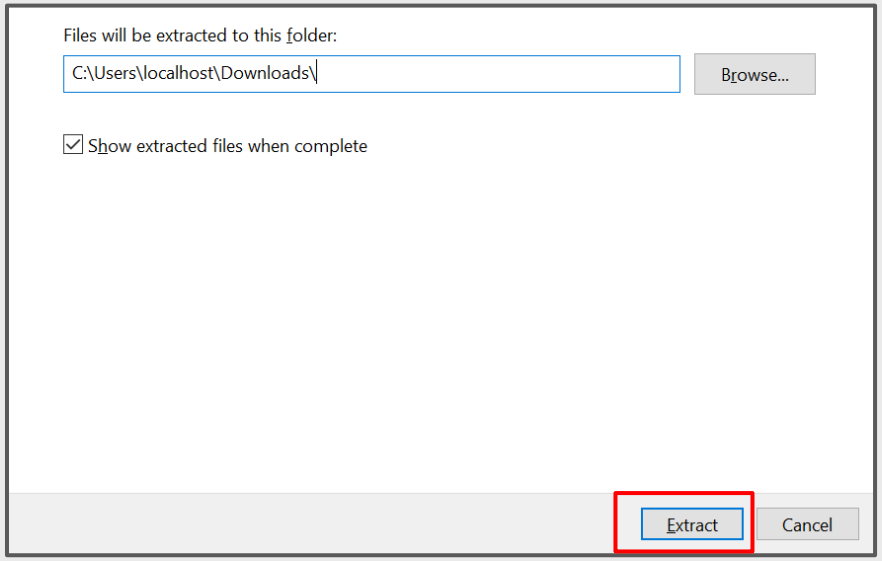


Open the Emojifier Starter Code (PC)

1. Open your Downloads folder.
2. Click the file.
3. Click **Extract**, then **Extract All**.
4. Delete the last part of the file name.
5. Click Extract.



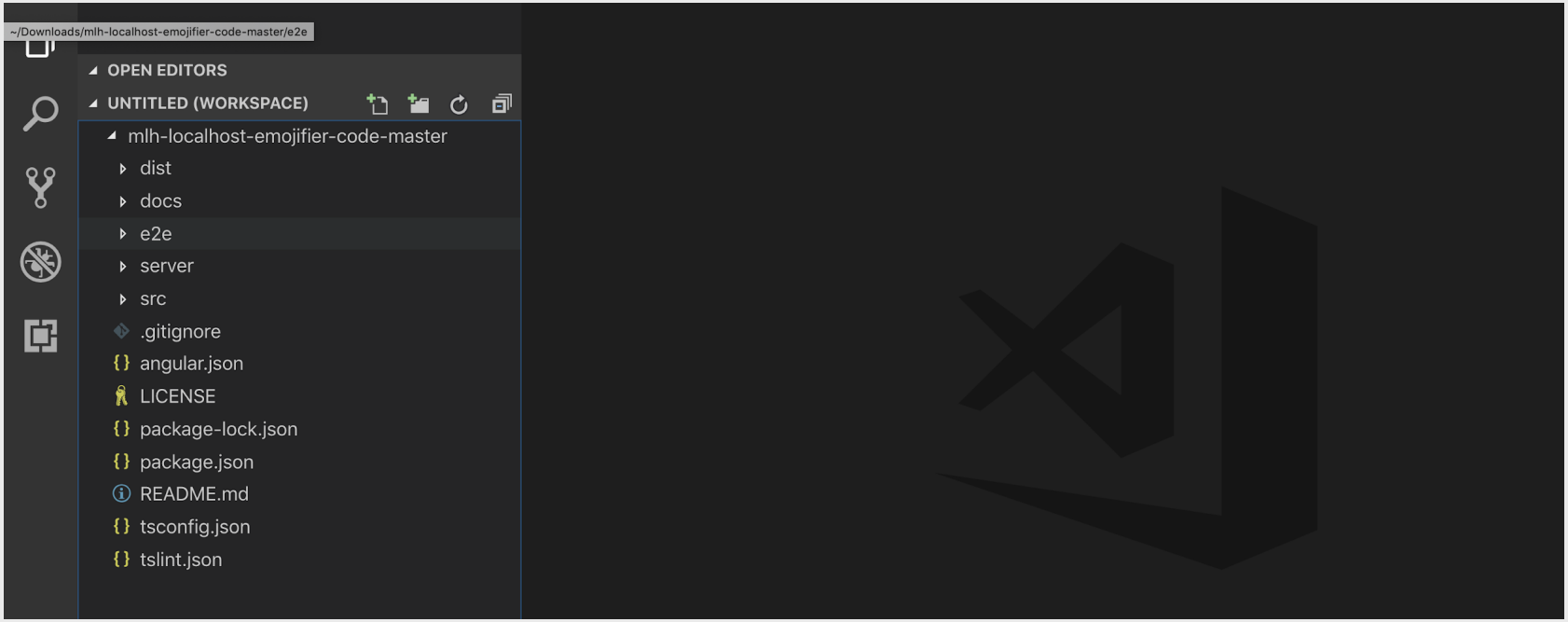




Open the Emojifier Starter Code

Open the project in the text editor of your choice. If you don't have a text editor installed, we recommend Visual Studio Code. Navigate to the URL below to install it.

http://mlhlocal.host/vscode

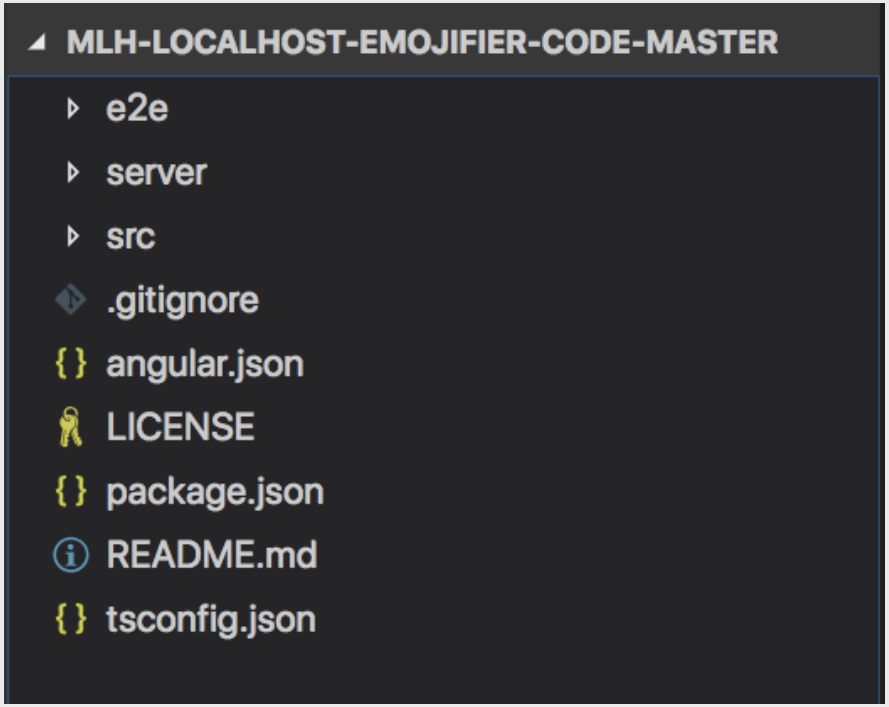


Let's take a short pause here and wait until everyone has completed the following:

1. Downloaded the code
2. Unzipped the code
3. Opened the project in a text editor.

Project Layout

A full stack application has lots of files! Let's explore the structure of this project.

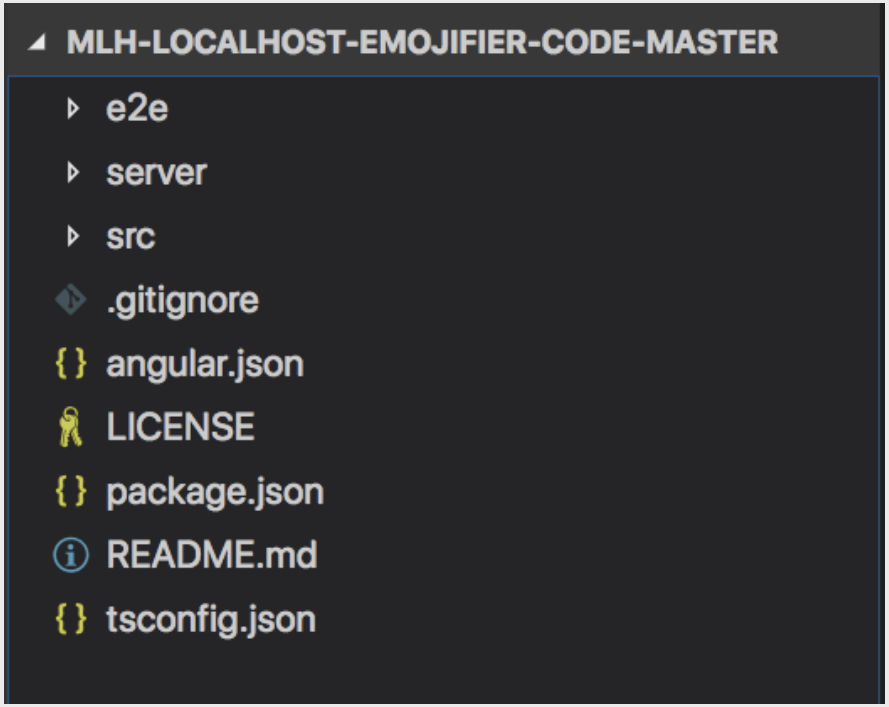


- **`e2e/`** contains “end-to-end” test files for our project. Not used in this project.

  In a typical full-stack project you would want to include some tests to verify your app behaves correctly.

Project Layout

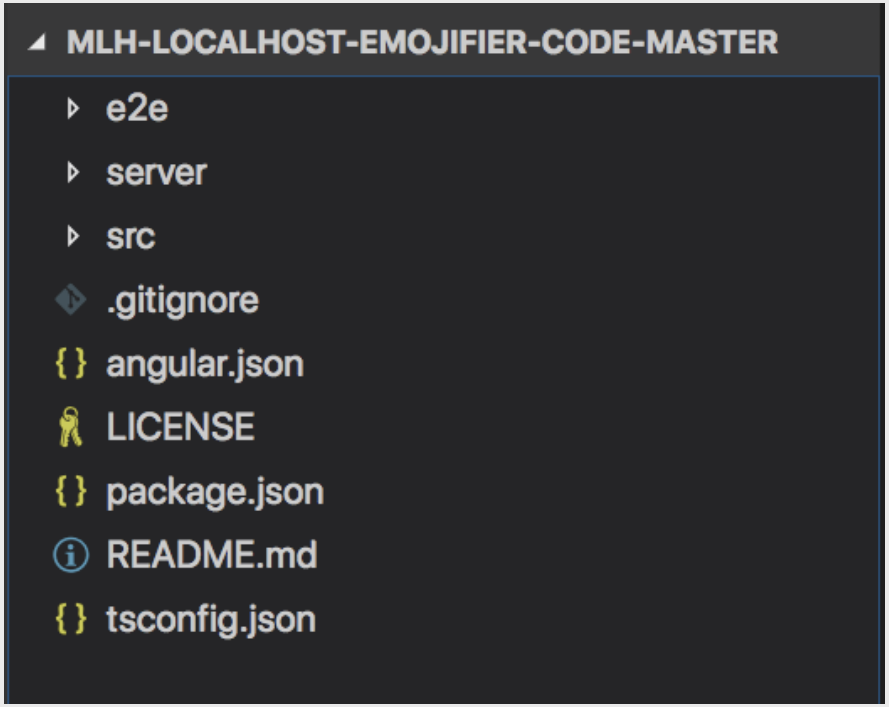
A full stack application has lots of files! Let's explore the structure of this project.



- The **`LICENSE`** file includes information about who owns the project and copying/use rules.
- The **`README.md`** file contains information about how to run the project.

Project Layout

A full stack application has lots of files! Let's explore the structure of this project.

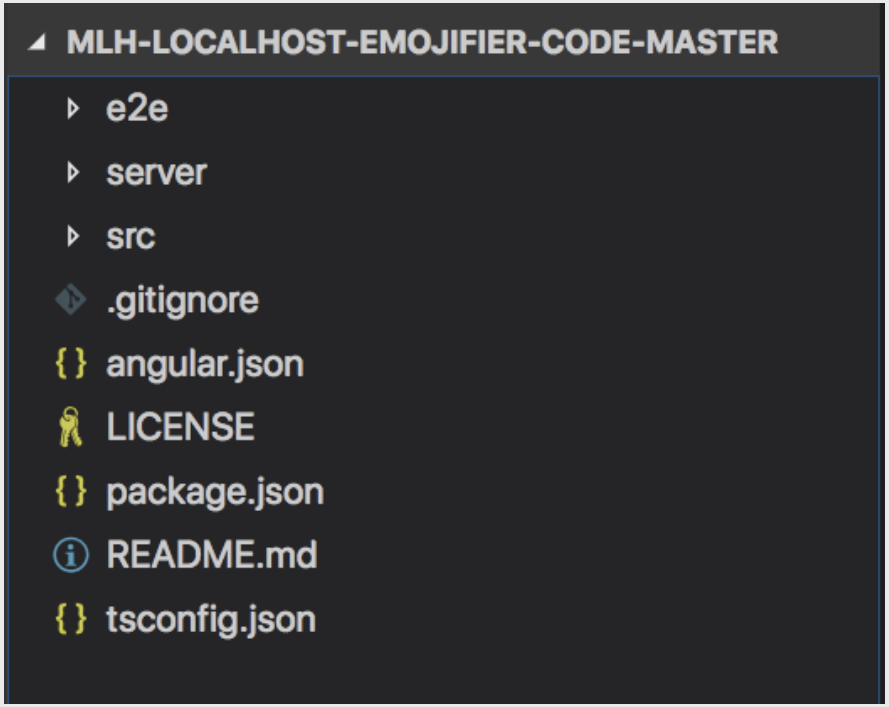


- **`angular.json`** info about our angular project.
- **`package.json`** includes all of the node modules that our project needs to run.

Both files define scripts for building and running our project.

Project Layout

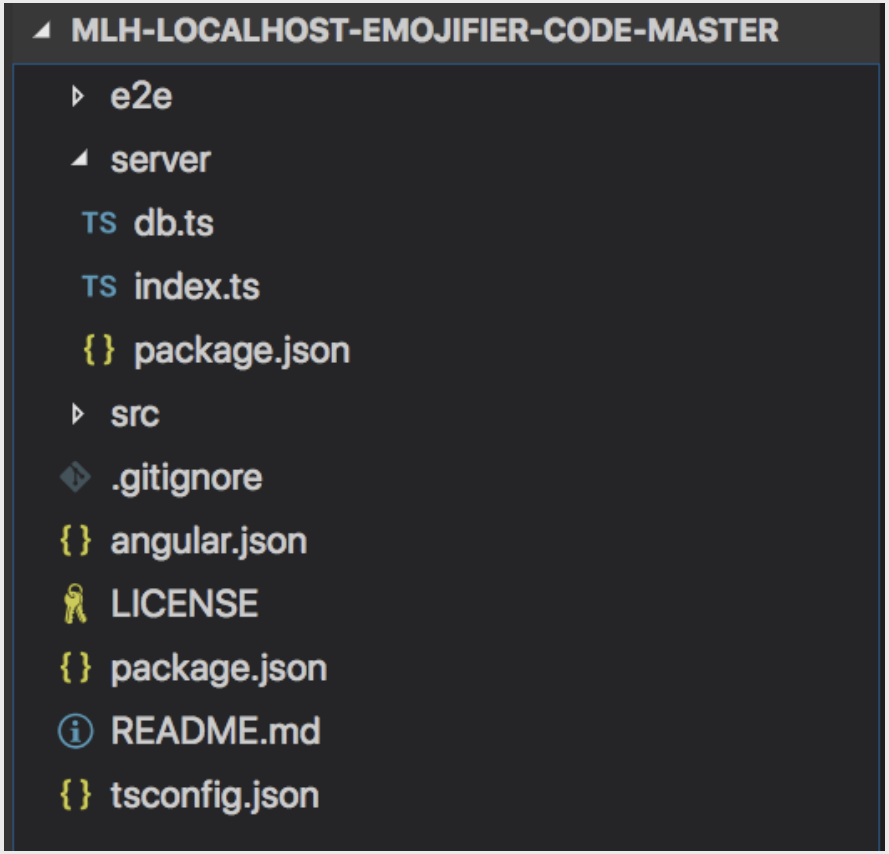
A full stack application has lots of files! Let's explore the structure of this project.



- **`tsconfig.json`** configures TypeScript, a superset of JavaScript that is used in Angular. We'll learn more about this later.

Project Layout

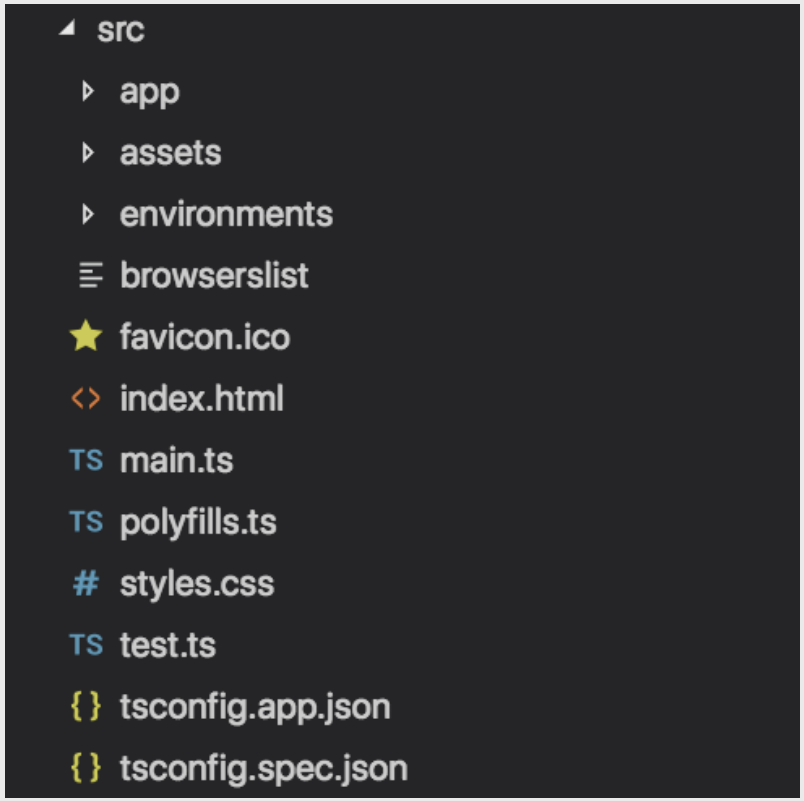
A full stack application has lots of files! Let's explore the structure of this project.



`server/` contains

- **`index.ts`** manages the back end, or server-side, of our application.
- **`db.ts`** manages the connection to MongoDB.

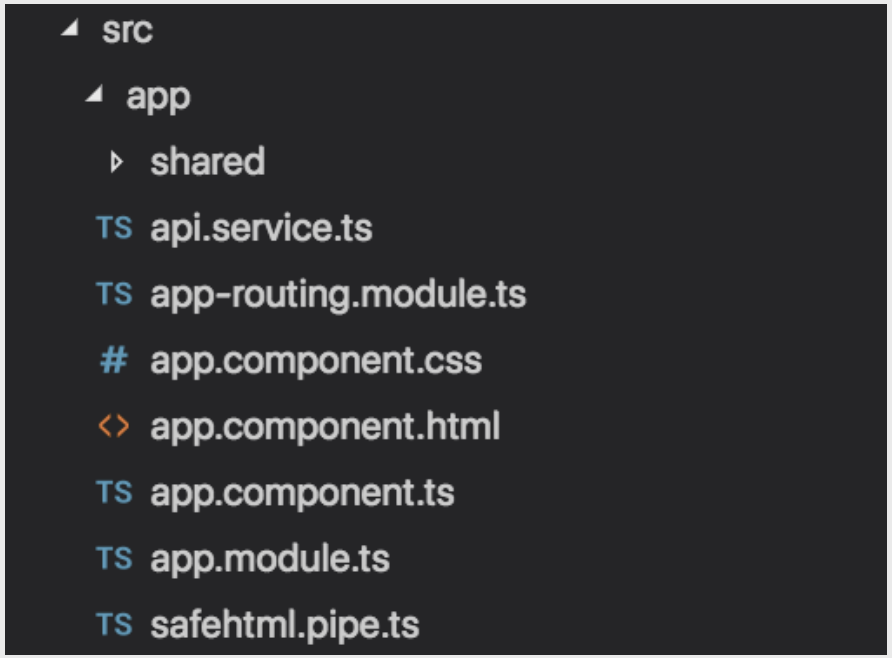
Project Layout src/



`src/` contains many files:

- **`assets`** fonts and images used in our project.
- **`environments`** files defining project behaviour depending if we're in development or production mode. Not used in this tutorial.

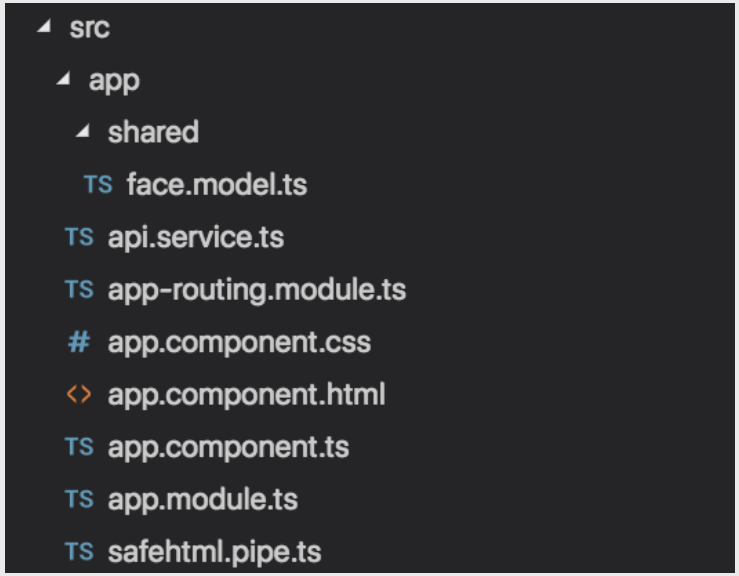
Project Layout src/app



`src/app/` contains the front-end of our project. You're likely familiar with CSS and HTML files.

- We'll be editing **`app.component.ts`** in this workshop – the file that adds the Emojified image to the page!
- You might be unfamiliar with the **`.ts`** ending. These are TypeScript files, and we'll learn what that means later.

Angular Concepts



`app.component.ts` imports from:

- **`face.model.ts`** defines the form of data we're getting from the Microsoft Face API
- **`safehtml.pipe.ts`** transforms a component attribute for display
- **`api.service.ts`** defines a service, code which doesn't modify any visuals, for getting the back-end API response

`app-routing.module.ts` isn't really used in this app. Used for managing multiple view states in complex apps.

Now that you understand all the components of this project, let's try the app!

Test the Project!

1. Open a terminal. Type `pwd` to make sure you are in the right directory. If you are not, `cd` into the directory where you installed the emojifier code.
2. Type `npm install` to install dependencies.
3. Enter the command below for your operating system.
4. Open your browser and navigate to **`localhost:4200`**

```
$ pwd
/Users/Localhost/[YOUR_DOWNLOAD_LOCATION]/mlh-localhost-emojifier-code-master

$ npm install

< lots of output >

# Windows command
$ npm run start-win

# Mac/Linux command
$ npm run start-mac

< lots of output >
```

Test the App!

5. You should see a web page like below. Enter the URL for an image. You can copy one from the README if you don't have one.

It doesn't work!

Let's find out why.

Table of Contents

As we explore the project, we're going to find lots of missing code that we'll need to replace!

Make `index.ts` send API requests

Our app isn't working. Let's open **`server/index.ts`** to find out why. There's a bunch of code missing, and we're going to write it! **`index.ts`** has a lot of setup code, but this request code will be our main focus:

```
19  app.post('/', (req, res) => {
20    const { imageUrl } = req.body;
21
22    const params = {
23      'returnFaceAttributes': 'emotion'
24    };
25
26    const options = {
27      url: uriBase,
28      qs: params,
29      body: '{"url": ' + '"' + imageUrl + '"}',
30      headers: {
31        'Content-Type': 'application/json',
32        'Ocp-Apim-Subscription-Key' : subscriptionKey
33      }
34    };
35
36    // TODO: Send Request to Face API
37    // TODO: Send Face API response to front-end
38    // TODO: Save Face API response to database
39  });
```

Get Image URL from front-end.

Setup the Face API request.

This is the code we're writing.

You need an API key from the Microsoft Face API for your app to work.

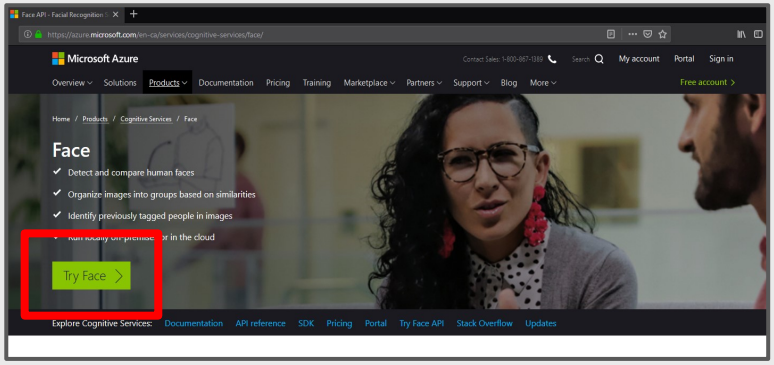
Let's get that now!

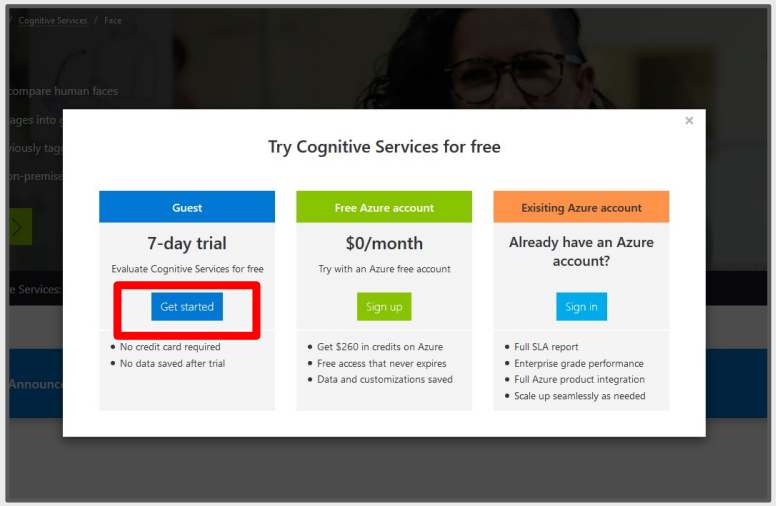
Get your Microsoft Face API key

We'll use the Microsoft Face API to analyze the emotions from images. Let's get an API key now.

1. Navigate to the URL below.
2. Select **Try Face**.
3. In the screen that opens, select **Get Started** under 7-day trial.

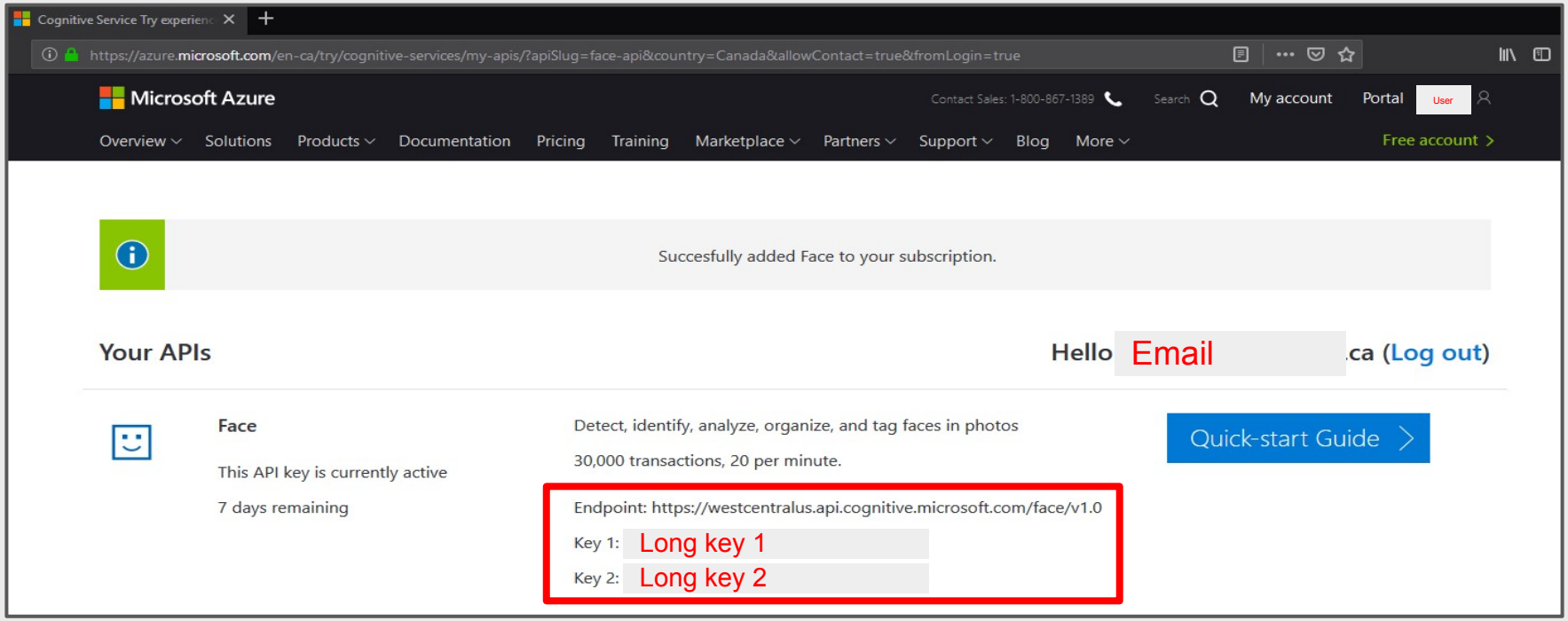
http://mlhlocal.host/face-api





Get your Microsoft Face API key

3. Save these keys and the Endpoint URL to a text file or leave this browser window open. We'll need them later.



Send API request

1. Verify that your endpoint URL matches the variable `uriBase`, around line 14. If not, replace it. Please also add `/detect/` to the end of the URI.

```
17 | // When you sign up for the Face API, your region might be different.
18 | const uriBase = 'https://westcentralus.api.cognitive.microsoft.com/face/
     v1.0/detect/';
```

2. Then, paste your API Key 1 on line 15.

```
20 | // When you sign up for the Face API, you'll be given a subscription key.
21 | const subscriptionKey = 'dontstealmykeypls';
```

Send API request

3. Add the following code, beginning on or around line 44.
   This code submits a `post` request to the Face API when a user clicks the `Emojify!` button. Then, it logs the response to the console.

```
36 |  request.post(options, (error, response, body) => {
37 |     console.log(body);
38 |  })
```

Send API response

4. Insert lines 38-39.

- **Line 38:** Set data-type to JSON, so the Front-End knows how to read it.
- **Line 39:** Send the API response `body` to the Front-End.

```
36 |  request.post(options, (error, response, body) => {
37 |    console.log(body);
38 |    res.setHeader('Content-Type', 'application/json');
39 |    res.send(body);
40 |  })
```

Test the App again!

5. If your app is still running, kill it with **CTRL C** in the terminal. Then, run **`npm start`** again. When you submit the image from the web page, you should see a response from the Face API printed in your console:

```
[SERVER]   body:
[SERVER]    '[{"faceId":"c31c2a14-1308-422d-82ae-be57014b6422","faceRectangle":{"top":70,"left":183,"width":60,"height":60},"fac
eAttributes":{"emotion":{"anger":0.0,"contempt":0.0,"disgust":0.0,"fear":0.0,"happiness":0.917,"neutral":0.001,"sadness":0.024,"
surprise":0.057}}}]' }
```

Depending on the image you used, the values of each emotion will be different. The app will choose the emotion with the greatest value.

We'll learn later in this workshop how we're going to use the response from the Face API!

Our Back-End is sending and receiving messages.

Now we need our Front-End to listen to updates!

Table of Contents

Code Review: `app.component.ts`

Open **`src/app/app.component.ts`** in your editor, which contains our app's the main components. Components are how Angular organizes groups of code which modify web page contents. Different sections of the same web site might be different components.

```
14  @Component({
15    selector: 'app-root',
16    templateUrl: './app.component.html',
17    styleUrls: ['./app.component.css'],
18  })
19
20  export class AppComponent {
21    constructor(private http: HttpClient) {}
22
```

- **Lines 14-18:** Use the `@Component` decorator to register **`AppComponent`** with Angular.

Key Terms

component: groups of code that modify web page contents.

decorator: wraps function or class calls to change behavior. In this case, we're telling Angular what other files this class depends on.

The Emoji Component: Types

```
14  @Component({
15    selector: 'app-root',
16    templateUrl: './app.component.html',
17    styleUrls: ['./app.component.css'],
18  })
19
20  export class AppComponent {
21    constructor(private http: HttpClient) {}
22
```

Key Term

types: ways of describing a variable to the code runner.

typescript: translated JavaScript that allows you to specify the type of a variable. This compiles to JavaScript.

- **Lines 20-21**: Initialize Component with **`http`** to communicate with the Back-End.
- **`http: HttpClient`** indicates the attribute **`http`** is of type **`HttpClient`**.

Types are ways of describing a variable to the code runner. This file looks like JavaScript, but is actually TypeScript, which is why we can have types. TypeScript is translated into JavaScript via a code reader/compiler, so it can run in the web browser.

More about types on the next slide!

Detour: Defining Types

JavaScript

JavaScript does not let you define types, so it lets you write any code you want.

```
var a = "aa";
var b = "bb";
var c = aa * bb; // you can't
    multiply text !?!?!
```

JavaScript would allow this, even if it's weird.

TypeScript

TypeScript lets you assign types to variables and will stop if you do something with the types that don't make sense.

```
var a: string = "aa";
var b: string = "bb";
var c = aa * bb;
```

TypeScript knows this isn't allowed.

Types let you:

- Find bugs in your code before you run it.
- Helps your editor make better code suggestions.

Types are sometimes hard to write and can slow down coding. When writing code, there's always a trade-off between quickly prototyping without types and adding types once structure is established.

The Emoji Component: Attributes

Let's return to **`app.component.ts`** to continue reviewing the code.

- **Lines 22-25:** Initialize Component attributes for HTML rendering + communication.

  We said earlier that components are like templates. Our template has the following attributes - `title`, `imageUrl`, and `htmlToAdd`. These are the elements we're going to add to our page.

  At the beginning of our component, we're setting up the default values. We can update these dynamically based on user interactions.

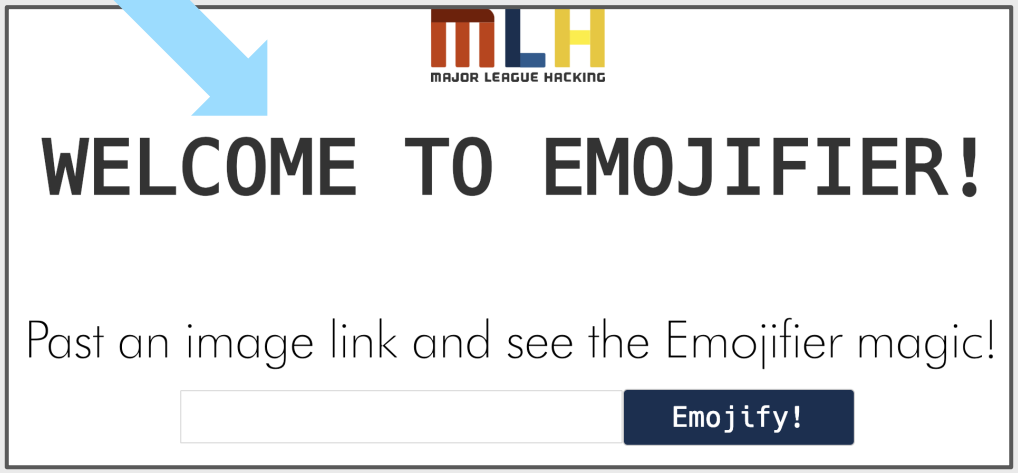
```
20  export class AppComponent {
21    constructor(private http: HttpClient) {}
22
23    title = 'emojifier';
24    uri = 'http://localhost:3000';
25    imageUrl = '';
26    htmlToAdd = ''
```

Attributes mapped to HTML templates

`app.component.ts` set up some attributes. Let's open **`app.component.html`** to see where these attributes are used and how they're inserted into the HTML.

Angular marks values to insert into HTML with **`{{ value_to_insert }}`** like inserting the title on line 26 of **`app.component.html`**.

```
25  <h2 style="font-family: 'Hack', sans-serif; font-weight: Bold; color: #333;
    margin: 4vh 2vw; padding: 0; font-size: 8vh;">
26      WELCOME TO {{ title }}!
27  </h2>
```



Attributes mapped to HTML templates

Remember, when you submit an image URL, that image is evaluated then added to the webpage. In app.component.ts, that's the imageUrl attribute. Let's see how that happens in app.component.html.

- **Line 43:** Inserting the image is more complicated. `*ngIf="!!imageUrl"` checks if **`imageUrl`** is defined and otherwise doesn't insert the HTML element.
- **Line 44:** To insert raw HTML, such as the emoji, the attribute `htmlToAdd` has to be filtered by `safehtml:'html'` as marked by the vertical line, called the "pipe" symbol.

```
42 |   <div style="position: relative">
43 |     <img src="{{ imageUrl }}" *ngIf="!!imageUrl"/>
44 |     <div [innerHtml]="htmlToAdd | safehtml:'html'"></div>
45 |   </div>
```

The sendFile method

Now that you understand how the app dynamically changes the Emojified image, let's return to **`app.component.ts`** to continue reviewing the code.
The `sendFile()` function is where all the front-end work gets done. We'll review the code it uses to:

1. Send the URL entered by the user to the Back-End for processing.
2. Read the face response from the Back-End.
3. Replace the face with an emoji.

```
19 |    sendFile(imageUrl) {
20 |      this.imageUrl = '';
21 |
22 |      const obj = {
23 |        imageUrl: imageUrl,
24 |      };
```

The sendFile method setup

- **Lines 19-24:** Setup processing request and response display.
- **Line 27:** Send image to Back-End for processing.
- **Lines 28-31:** Display error if request fails.

```
19 |     sendFile(imageUrl) {
20 |       this.imageUrl = '';
21 |
22 |       const obj = {
23 |         imageUrl: imageUrl,
24 |       };
25 |
26 |       this.htmlToAdd = '';
27 |       this.apiService.getImageData(imageUrl, (res) => {
28 |         if (res.error) {
29 |           this.htmlToAdd += `<span style="color: red;">${res.error.message}</
30 |   span>`;
31 |           return;
          }
```

Write some code!

1. Let's add a new type for the Face API response, `FaceModel`, to the code. This is what will add the image to the page and superimpose the Emoji on top of it.
2. In the file **`src/app/shared/face.model.ts`**, add the following code:

```
1 |   export type FaceModel = {
2 |
3 |    };
```

The sendFile method setup

3. Inside `FaceModel`, add an object called `faceRectangle` that has four properties. These properties are how we'll tell our app where the image's face is. This is how it will know where to put the emoji.

```
export type FaceModel = {
    faceRectangle: {
        height: number;
        width: number;
        left: number;
        top: number;
    };
};
```

The sendFile method setup

4. Below `faceRectangle`, add the following code.

```
    export type FaceModel = {
     faceRectangle: {
         height: number;
         width: number;
         left: number;
         top: number;
     };
     faceAttributes: {
         emotion: object;
     };
    };
```

Compare FaceModel to API Response

Do you remember earlier when you added your API key to **`index.ts`** and submitted your first image to be analyzed? The Face API returned a response in your terminal. Below is a formatted version of that response. Compare that to the type we just created. What do you notice?

```
"faceId":"81791d31-e0e7-4aff-a577-d05c9f393b00",
"faceRectangle":{
   "top":80,
   "left":110,
   "width":136,
   "height":136
},
"faceAttributes":{
   "emotion":{
      "anger":0.0,
      "contempt":0.0,
      "disgust":0.0,
      "fear":0.0,
      "happiness":0.0,
      "neutral":0.0,
      "sadness":0.0,
      "surprise":1.0
   }
```

```
export type FaceModel =
{
 faceRectangle: {
     height: number;
     width: number;
     left: number;
     top: number;
 };
 faceAttributes: {
     emotion: object;
 };
};
```

Compare FaceModel to API Response

The API returns **`faceRectangle`** which tells us where the face is on the image we submitted. It also returns **`faceAttributes`**, specifically emotion, which we used to figure out which emoji to superimpose. In the next few steps, we'll add this information to the **`htmlToAdd`** attribute so that we can add this image and emoji to our component!

```
"faceId":"81791d31-e0e7-4aff-a577-d05c9f393b00",
"faceRectangle":{
   "top":80,
   "left":110,
   "width":136,
   "height":136
},
"faceAttributes":{
   "emotion":{
      "anger":0.0,
      "contempt":0.0,
      "disgust":0.0,
      "fear":0.0,
      "happiness":0.0,
      "neutral":0.0,
      "sadness":0.0,
      "surprise":1.0
   }
```

```
export type FaceModel =
{
 faceRectangle: {
      height: number;
      width: number;
      left: number;
      top: number;
 };
 faceAttributes: {
      emotion: object;
 };
};
```

You just created a type! Awesome.
Let's explore the function further.

The sendFile processing face

1. Return to **app.component.ts**.
2. We have a block that is commented out. Uncomment **lines 32** to **48** by deleting the double slashes at the beginning of each line. You can also do this by highlight the block and typing **COMMAND /** on a Mac or **CTRL /** on PC.

```
32        // res.forEach((face: FaceModel) => {
33        //  const { faceRectangle, faceAttributes } = face;
34        //  const { height, width, left, top } = faceRectangle;
35        //
36        //  // TODO: Add image to website here
37        //
38        //  const { emotion } = faceAttributes;
39        //  let mainEmotion = undefined;
40        //
41        //  Object.keys(emotion).forEach(key => {
42        //    if(!mainEmotion || emotion[key] > emotion[mainEmotion]) {
43        //      mainEmotion = key
44        //    }
45        //  });
46        //
47        //  // TODO: Add emoji to image here
48        // })
```

The sendFile processing face

Now your code looks like this. Here's what it does:

- **Line 32:** Iterate through response (there will be one for each face)
- **Lines 33-34:** Extract response pieces for displaying emotion
- **Lines 38-45:** Get the detected emotion from the result
- **Line 47:** Actually display the emoji, which we need to implement!

```
32      res.forEach((face: FaceModel) => {
33        const { faceRectangle, faceAttributes } = face;
34        const { height, width, left, top } = faceRectangle;
35
36        // TODO: Add image to website here
37
38        const { emotion } = faceAttributes;
39        let mainEmotion;
40
41        Object.keys(emotion).forEach(key => {
42          if(!mainEmotion || emotion[key] > emotion[mainEmotion]) {
43            mainEmotion = key
44          }
45        });
46
47        // TODO: Add emoji to image here
48      })
```

Note: Lines 55, 56, and 60 are using a JavaScript convention called destructuring. You can learn more about these at **`mlhlocal.host/destructure`**.

Ok! Let's add the code that adds the original image to the webpage, then the code to add the emoji!

Updating attributes to display

1. Add the following code on lines 34 and 35.
    - **Line 34:** Set the position of the emoji to be the Face API face coordinates.
    - **Line 35:** Set the image to be the image the user submitted.

```
34 | const style = `position:absolute; height:${height}px; width:${width}px;
       left:${left}px; top:${top}px`;
35 | this.imageUrl = imageUrl;
```

Updating attributes to display

2. Add the code below on line 49. This code sets the image to be the given emotion and emoji in the `/assets/` folder.

```
49 | this.htmlToAdd += `<img class="emoji" style="${style}" src="/assets/${mainEmotion}.png"/
>`;
```

Let's look at the code below to understand how this works.

- `emotion` contains values for how much the face appears to show anger, contempt, disgust, fear, happiness, neutrality, sadness, or surprise.
- Iterate over the values for each of these emotions to find the greatest one.
- Then we set `mainEmotion` to be the greatest value and grab that emoji from the `assets` directory and put it in the image tag above!

```
38          const { emotion } = faceAttributes;
39          let mainEmotion;
40
41          Object.keys(emotion).forEach(key => {
42            if(!mainEmotion || emotion[key] > emotion[mainEmotion]) {
43              mainEmotion = key
44            }
45          });
```

Test the App!

MAJOR LEAGUE HACKING

WELCOME TO EMOJIFIER!

Past an image link and see the Emojifier magic!

https://image.shutterstock Emojify!

shutterstock.com • 583558969

The Face API results are being translated into Emoji and displayed!

Let's save the results into our database.

Table of Contents

Save face to MongoDB

1. Return to **`server/index.ts`**
2. Add the following code in the request, directly below **`res.send(body)`**:

- **Line 43:** Check to see if the response from the Face API is 200.
- **Line 44:** If it is, save the response to the Mongo Database

```
39 |   request.post(options, (request, response, body) => {
...    …
43 |       if(response.statusCode == "200"){
44 |         saveFace(imageUrl, JSON.stringify(body));
45 |       }
46 |   }
```

Get correct MongoDB collection

3. Switch to **`server/db.ts`** where the **`saveFace`** function is implemented.
4. Add the following code under the `MongoClient.connect()` function.

- **Line 8:** Throws an error if connecting to MongoDB failed
- **Line 9**: Connect to the database called `dbName` (`"emojifier"`)
- **Line 10**: Create a collection of documents called `"faces"`.

```
7  |  MongoClient.connect(mongoURL, function(err, db){
8  |    if (err) throw err;
9  |    var dbo = db.db(dbName);
10 |    dbo.createCollection("faces");
11 |    // ... more stuff to follow
12 |  });
```

Save face to MongoDB

5. Then, add the following code.

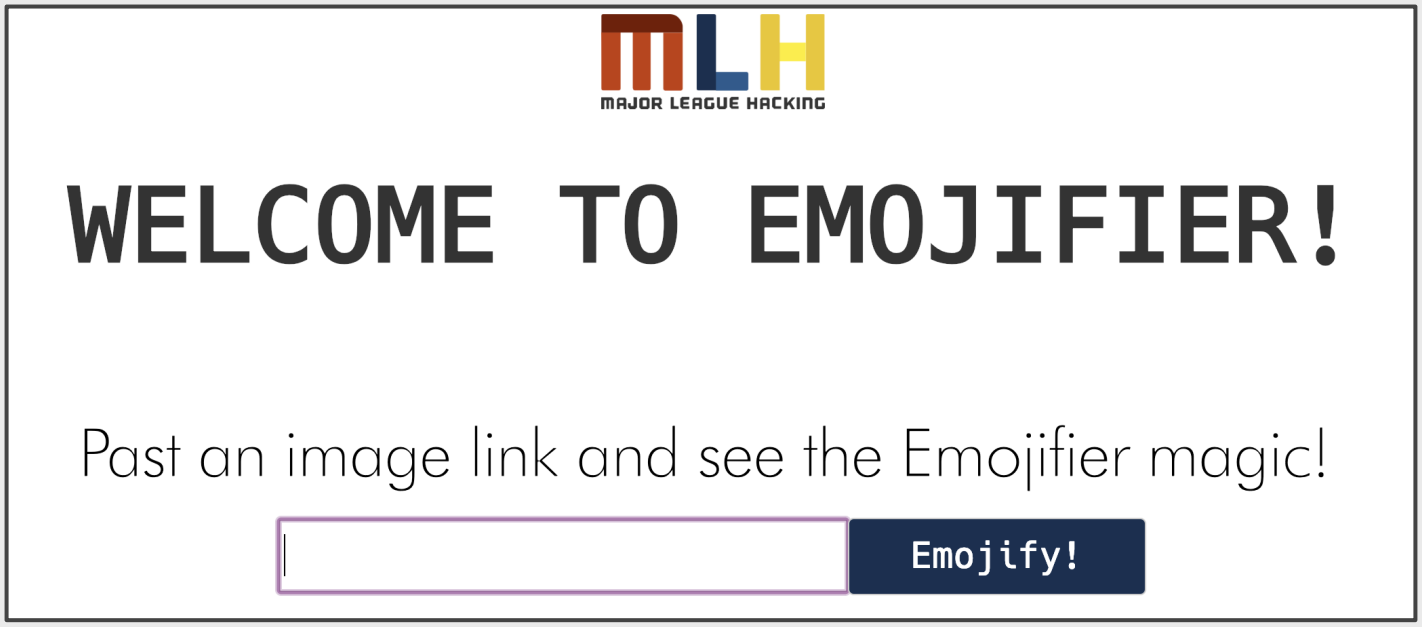
- **Line 11:** Create an object that contains the face data. This will be stored in MongoDB
- **Line 12:** Use the **`insertOne()`** method to add the face object you just created to the database.
- **Line 13:** Throw an error if one occurs.
- **Line 14:** Log to the console that we have added an entry to the database.
- **Line 15:** Close the database.
- **Line 16:** Close the **`insertOne()`** method.

```
11 |   const myobjFace = { imageUrl: imageUrl, faceAttributes: data };
12 |   dbo.collection("faces").insertOne(myobjFace, function(err, res) {
13 |     if (err) throw err;
14 |     console.log("1 register inserted");
15 |     db.close();
16 |   });
```

Test the App again!

You should see something like the output below after submitting the image. The image URL and the returned attributes from the Face API.

```
[2] i 「wdm」: Compiled successfully.
[SERVER] [{"faceId":"6292bacf-46c1-40f6-aa33-cedf2e0a815a","faceRectangle":{"top":193,"left":565,"width":887,"height":886},"face
Attributes":{"emotion":{"anger":1.0,"contempt":0.0,"disgust":0.0,"fear":0.0,"happiness":0.0,"neutral":0.0,"sadness":0.0,"surpris
e":0.0}}}]
[SERVER] (node:11104) DeprecationWarning: current URL string parser is deprecated, and will be removed in a future version. To u
se the new parser, pass option { useNewUrlParser: true } to MongoClient.connect.
[SERVER] 1 register inserted
```



Congratulations!

We've finished a full MEAN stack app with lots of room for experimentation and exploration!

Table of Contents

Let's recap quickly...

1. The MEAN stack is a combination of technologies for web apps.
2. Angular is a framework for building interactive components using HTML and TypeScript.
3. MongoDB saves data.
4. Node and Express glues components together.

What did you learn today?

We created a fun quiz to test your knowledge and see what you learned from this workshop.

http://mlhlocal.host/quiz

Table of Contents

Keep Learning: Practice Problems for Later

Extra Practice Problem 1:

Show the last submitted image on the home page.

Extra Practice Problem 2:

Extract the MongoDB and localhost URI to environment variables for easier deployment.

Learning shouldn't stop when the workshop ends...

Check your email for access to:

- These workshop slides
- Practice problems to keep learning
- Deeper dives into key topics
- Instructions to join the community
- More opportunities from MLH!


Workshop
Intro to the MEAN Stack
Powered by BlackRock
MLH localhost
BLACKROCK®