Project 1: JavaScript Full Stack Nanodegree

Image Processing API

Project Overview

You will be building an API that can be used in two different ways. As a simple placeholder API, the first allows you to place images into your frontend with the size set via URL parameters (and additional stylization if you choose) for rapid prototyping. The second use case is as a library to serve properly scaled versions of your images to the front end to reduce page load size. Rather than needing to resize and upload multiple copies of the same image to be used throughout your site, the API you create will handle resizing and serving stored images for you.

You will set up your Node.js project from scratch, install all dependencies, and write all necessary configurations to make your dependencies work together. Even though the project being built is quite simple, the way you set up this project will be scalable enough to move to an enterprise-level solution in the future. Imagine needing to process hundreds of images with multiple thumbnail sizes for an eCommerce solution. This project provides the building blocks for that level of functionality.

In addition to setting up and creating the functionality, you will use industry best practices to ensure that your code is as scalable as your architecture. Using TypeScript, unit testing, linting, and formatting, you will write code that is easy to read, maintainable, less error-prone, and easier to debug. At the enterprise level, it becomes possible that hundreds of people may need to interact with your code, so being able to work within standards is imperative to your success.

If you choose to work on your own system, you can download the images from Github:

- **https://github.com/udacity/cd0292-building-a-server-project-starter**

Project Summary

This project aims to give you a real-world scenario in which you would read and write to your disk via a Node.js express server rather than a database. The project you create serves two purposes: to prepare you for setting up scalable code and architecture for real-world projects and to tie together some of the most popular middleware and utilities found in Node.js projects.

This project barely touches the surface of what is possible but will prove your ability to use what you've learned in real-world scenarios.
For this project, refactor and test as much as possible while you are building. Since you are using TypeScript and an unfamiliar library, it is sometimes easier to write and build in plain JS to see what your functions return; remember your submission needs to be in TypeScript. As your skills improve, typing in TypeScript will feel more intuitive. Make sure to remove any debugging code from your final submission.

What Will I learn?

The API you create presents your first opportunity to pull together the skills you learned through the course and tie them together in a commonly used application. Besides solidifying your skills, you'll also have the opportunity to problem-solve and work with the documentation of a popular image processing utility.
The following are just some of the questions that you'll experience along the way:

- What's the ideal workflow?
- How should I structure my project?
- How do I want to write my asynchronous TypeScript?
- How many functions do I need to complete this task?
- What types of things should I test for?

There's no single correct answer to each question. While building this project, working with peers, and getting feedback from the project reviewer -- you will naturally develop your own answers to these questions!

Project: Image Processing API

Setup and Architecture

Success Criteria	**Specifications**
Set up a project structure that promotes scalability	● Source code is kept separate from compiled code. ● All tests should be contained in their own folder. ● Separate modules are created for any processing.

Set up an npm project	● Package.json should contain both devDependencies, and dependencies. ● Scripts should be created for testing, linting/prettier, starting the server, and compiling TS. ● Build script should run without error. Note: Ensure that dependencies only include libraries that are necessary for your code to function after transpilation. devDependencies, on the other hand, are packages that are only needed for development.

Functionality

Success Criteria	**Specifications**
Add and use Express to a node.js project	● Start script should run without error ● Provided endpoint should open in the browser with status 200
Follow middleware documentation to use middleware to create an API	● Accessing the provided URL with image information should successfully resize an image and save it to disk on first access, then pull it from disk on subsequent access attempts. ● An error message should be provided to the user when an image has failed to process or does not exist. Tip: Don't forget to handle the common error scenarios: ● Missing filename, height, or width. ● Invalid Input for filename. ● Invalid Input for height or width.

Code Quality

Success Criteria	Specifications
Write relevant unit tests with Jasmine and SuperTest to improve code quality and refactoring	● Test script runs, and all tests created pass. ● There is at least 1 test per endpoint and at least one test for image processing. Note: ● Both tests, including those conducted through API requests and those performed by directly invoking the function, should be implemented. ● Don't forget the image processing test.
Utilize TypeScript to avoid errors and improve maintainability	● All code in the SRC folder should use the .ts filetype. ● Functions should include typed parameters and return types and not use the `any` type. ● Import and Export used for modules. ● Build script should successfully compile TS to JS.
Write well-formatted linted code	Prettier and Lint scripts should run without producing any error messages.

Suggestions to Make Your Project Stand Out

- Add additional processing to accept and output other image formats than JPG.
- Modify the thumbnail filename to include the image size to allow for multiple sizes of the same image.
- Further explore the options in the Sharp module and add additional processing options.
- Add in logging to record when images are processed or accessed.
- Create a front-end for uploading more images to the full-size directory.
- Create a front-end that displays a thumbnail directory.
- Create a front-end that allows for the selection of how to process a selected image.

here is a walkthrough to get you up and running!

- **Initialize your project.** Add the dependencies required for this project, including Express, TypeScript, Jasmine, Eslint, and Prettier. Complete your package.json file.
    - Where should your dependencies be placed?
    - What scripts should you create to take advantage of the dependencies you've added?
    - Are there other dependencies you would like to add or know you will need to improve your workflow?
- **Set up your project structure.** Create folders and files for what you anticipate you will need for the project.
    - How do you plan to keep your source code and build code separately?
    - Where will you keep your tests?
    - How do you plan to name your routes? Utilities?
- **Configure your middleware and dependencies.** You have quite a few dependencies that all need to work together. Sometimes it's easiest to write some simple js functions to test that all of your dependencies work together before you begin adding any functionality.
    - Does your TypeScript compile?
    - Do your Eslint and Prettier scripts work?
    - Are you able to write and pass a basic unit test?
- **Set up your server and create an API endpoint.** Even though this application is fairly straightforward, you still want to set it up in a scalable way. How can you set up your server and route so that your project remains scalable? Only one endpoint is required. It's best to create this and test that it is working before you move on.
- **Install Sharp and configure the endpoint.** Documentation for Sharp can be found **https://www.npmjs.com/package/sharp**

- . It is required that you create a separate module for your processing functionality and import it into your route. It is only required that you add resizing, but you may add additional processing if you choose to extend your application. It is also only required that you work with jpg files, so keep that in mind if you choose to use your own images and they are other formats.
    - Pay close attention to if you need to use asynchronous code or not. If you do, make sure you stay consistent throughout your application.
    - There is limited information on using Sharp with TypeScript, but don't let that be a blocker. Think about what type the Sharp constructor would return. Have a look at the **https://sharp.pixelplumbing.com/api-constructor**
    - and the examples it provides.
- **Write your tests.** If you haven't already been writing unit tests, now would be the time to start. Think about what you should test? At a minimum, you should have at least one test for your endpoint and at least one test for your image processing, but there are many different tests you could create.
- **Add caching.** Add caching to your application so that repeated requests to your endpoint use pre-stored images rather than regenerating a new image each time.

- **Test, Debug, and Refactor.** Think of edge-cases for your project and how someone may access your project. Should they get different error messages based on what's wrong? Make certain that your user isn't left in the dark when something goes wrong.
    - Do all of your tests still pass?
    - Does stopping and restarting your server cause any issues?
    - Does your user get feedback when something goes wrong?
    - Is your code easy to follow? Comments?
    - Is your API production-ready?
- **Build, Document, and Submit.** If everything else has gone well, you should be able to compile your typescript and start up your production server to test that everything still works as expected. Make sure you've provided all necessary information in your readme file, so your reviewer knows how to test your API. If everything works and your documentation is complete, you're ready to submit! Congratulations!

Version Control

Although not a requirement, we recommend using Git from the very beginning if you choose to build on your local environment or use the provided workspace. Make sure to commit often and to use well-formatted commit messages that conform to our
https://udacity.github.io/git-styleguide/
.
Git is installed within the workspace. Using git and pushing to a repository is the best way to back up and get a copy of your work.

Udacity Style Guides

Although Eslint and Prettier will handle most of your formatting needs, you should write your code and markup to meet the specifications provided in these style guides:

- **http://udacity.github.io/frontend-nanodegree-styleguide/css.html**
-
- **http://udacity.github.io/frontend-nanodegree-styleguide/index.html**
-
- **http://udacity.github.io/frontend-nanodegree-styleguide/javascript.html**
-

A note on plagiarism: Viewing someone else's code to get a general idea of implementation, then putting it away and starting to write your own code from scratch is okay. Please **do not copy someone's code**, in whole or in part. For further details, check out
https://udacity.zendesk.com/hc/en-us/sections/360000345231-Plagiarism
.