# Contributing to `pdf-lib`

:+1::tada: First off, thanks for taking the time to contribute! :tada::+1:

`pdf-lib` is a JavaScript library designed to create and modify PDF files in any JavaScript environment. We welcome contributions from the open source community! Please read through this document to learn how to setup and run the project on your machine. If you have any questions or run into trouble, please [create an issue](https://github.com/Hopding/pdf-lib/issues/new).

### Table Of Contents

- [Local Setup and Prerequisites](#local-setup-and-prerequisites)

- [Running the Unit Tests](#running-the-unit-tests)

- [Running the Integration Tests](#running-the-integration-tests)

- [Using the Scratchpad](#using-the-scratchpad)

- [Using the VSCode Debugger](#using-the-vscode-debugger)

- [Generating a Flamegraph](#generating-a-flamegraph)

- [Compiling the Project](#compiling-the-project)

- [Running the Linter](#running-the-linter)

- [Running the Type Checker](#running-the-type-checker)

- [Debugging Tips](#debugging-tips)

## Local Setup And Prerequisites

You can develop `pdf-lib` on Windows, Mac, or Linux machines. While most of the original code was developed on Macs, care has been taken to ensure that all scripts and commands needed for development are platform independent. (If you find anything that doesn't work on your machine/platform, please [create an issue](https://github.com/Hopding/pdf-lib/issues/new) or submit a PR!)

In order to work on `pdf-lib`, please ensure you have installed the following:

- \*\*Node.js\*\* provides the runtime needed to run this project. ([Installation instructions](https://nodejs.org/en/download/) - need `v9.0.0` or greater).

- \*\*Yarn\*\* is the package manager used for this project. ([Installation instructions](https://yarnpkg.com/en/docs/install) - need `v1.12.0` or greater).

- \*\*Git\*\* is the SCM used for this project. ([Installation instructions](https://git-scm.com/downloads) - need `2.17.2` or greater)

Next you'll need to clone the project:

```

git clone https://github.com/Hopding/pdf-lib.git

cd pdf-lib

```

After cloning the project, you'll need to install the dependencies. All dependencies are managed within the `package.json` file. This means all you have to do is run:

```

yarn install

```

If you don't see any errors or warnings, then everything should have worked correctly. The next thing you'll want to do is [run the unit tests](#running-the-unit-tests) to verify that everything is is good shape.

## Running the Unit Tests

We use the [Jest](https://jestjs.io/) framework to write unit tests for `pdf-lib`. All unit tests are kept in the [`tests`](./tests) directory.

To run the unit tests, execute the following:

```

yarn test

```

This should output something like:

```

yarn run v1.16.0

$ jest --config jest.json --runInBand

PASS tests/api/PDFDocument.spec.ts (13.238s)

PASS tests/core/parser/PDFObjectParser.spec.ts

PASS tests/core/parser/PDFParser.spec.ts

...

PASS tests/core/document/PDFTrailer.spec.ts

PASS tests/core/streams/AsciiHexStream.spec.ts

Test Suites: 44 passed, 44 total

Tests: 380 passed, 380 total

Snapshots: 0 total

Time: 22.975s, estimated 39s

Ran all test suites.

? Done in 23.66s.

```

Hopefully you see that all the tests passed! But if you see errors or warnings, then something must be wrong with your setup. Please ensure you've following the installation steps outlined in the [local setup and prerequisites section](#local-setup-and-prerequisites). If you still can't get the tests running after following these steps, then please [create an issue](https://github.com/Hopding/pdf-lib/issues/new) explaining the problem you're having.

## Running the Integration Tests

In addition to unit tests, we maintain a suite of integration tests for 3 different JavaScript environments. All integration tests are kept in the [`apps/`](./apps) directory. The goal of these tests is to ensure the project as a whole works correctly, whereas the unit tests ensure individual classes and functions work correctly. The integration tests take longer to run than the unit tests. They also require manual inspection of the PDFs they create to make sure nothing is broken, whereas the unit tests are entirely automatic (they pass/fail without human input).

> \*\*Make sure to [compile the code](#compiling-the-project) before running these tests\*\*

There are integration tests for Node, Deno, browser, and React Native environments:

- To run the tests for Node:

```

yarn apps:node

# Follow the prompts in your terminal

```

- To run the tests for Deno:

```

yarn apps:deno

# Follow the prompts in your terminal

```

- To run the tests for the browser:

```

yarn apps:web

# Open http://localhost:8080/apps/web/test1.html in your browser

```

- To run the tests for React Native (iOS):

```

yarn apps:rn:ios

# Tap through the tests in your simulator

```

- To run the tests for React Native (Android):

```

yarn apps:rn:android

adb reverse tcp:8080 tcp:8080

# Tap through the tests in your simulator

```

## Using the Scratchpad

The scratchpad is a handy tool for testing your code changes. It serves as an entrypoint to the code contained in [`src/`](./src). There are two steps required to use the scratchpad:

1. Start the TypeScript compiler:

```

yarn scratchpad:start

```

Note that you must leave this server running. It will detect code changes

as you make them and automatically recompile the code.

2. Run the [`scratchpad/index.ts`](./scratchpad/index.ts) file:

```

yarn scratchpad:run

```

This will compile and execute the code in your scratchpad file. If you

(1) create a PDF in the scratchpad, (2) save it to the file system, and (3) pass its file path to [`openPdf`](./scratchpad/open.ts#L11) function, then running this command will automatically open the file in your viewer of choice.

## Using the VSCode Debugger

You can use the VSCode debugger to run your scratchpad file. This can be a very powerful tool for testing and debugging the code. There are two steps required to use the debugger:

1. Start the TypeScript compiler:

```

yarn scratchpad:start

```

Note that you must leave this server running. It will detect code changes

as you make them and automatically recompile the code.

2. Run the [`scratchpad/index.ts`](./scratchpad/index.ts) file in the debugger

by clicking the `Start Debugging` button (or use the `F5` keyboard

shortcut).

## Generating a Flamegraph

Flamegraphs are incredibly useful visual tools for troubleshooting performance issues. You can generate a flamegraph using the [scratchpad file](#using-the-scratchpad) as an entrypoint by running `yarn scratchpad:flame` (note that you must have the TypeScript compiler running, as explained above). This will run the scratchpad file, generate a flamegraph of its execution, and automatically open it in your browser.

## Compiling the Project

For most development, manual compilation isn't necessary. The scratchpad and unit tests are usually all you need to test your code changes. But manual compilation \_is\_ necessary prior to running the integration tests or releasing a new version of the code to NPM.

Compiling the project will produce 4 artifacts:

- \*\*`compiled/cjs`\*\* - a directory containing a CommonJS version of the project (uses `require` instead of `import`). This folder contains `.js` and [`.d.ts`](https://stackoverflow.com/a/21247316) files, rather than the `.ts` files that the project source is written in.

- \*\*`compiled/es`\*\* - a directory containing an ES2015 version of the project (uses `import` instead of `require`). This folder contains `.js` and [`.d.ts`](https://stackoverflow.com/a/21247316) files, rather than the `.ts` files that the project source is written in.

- \*\*`compiled/dist/pdf-lib.js`\*\* - a single JavaScript file containing a [UMD](https://www.davidbcalhoun.com/2014/what-is-amd-commonjs-and-umd/) version of the project.

- \*\*`compiled/dist/pdf-lib.min.js`\*\* - a single JavaScript file containing a minified [UMD](https://www.davidbcalhoun.com/2014/what-is-amd-commonjs-and-umd/) version of the project.

To compile the project, execute the following:

```

yarn build

```

This should output something like the following:

```

yarn run v1.16.0

$ yarn build:cjs && yarn build:es && yarn build:umd && yarn build:umd:min

$ ttsc --module commonjs --outDir cjs

$ ttsc --module ES2015 --outDir es

$ rollup --config rollup.config.js --file dist/pdf-lib.js

es/index.js ? dist/pdf-lib.js...

created dist/pdf-lib.js in 1.5s

$ rollup --config rollup.config.js --file dist/pdf-lib.min.js --environment MINIFY

es/index.js ? dist/pdf-lib.min.js...

created dist/pdf-lib.min.js in 4s

? Done in 17.34s.

```

The compiled artifacts will be located in the `cjs/`, `es/`, and `dist/` directories.

## Running the Linter

We use two linters to keep `pdf-lib`'s source code clean, tidy, and consistent:

- [\*\*TSLint\*\*](https://palantir.github.io/tslint/)

- [\*\*Prettier\*\*](https://prettier.io/)

It is recommended that you setup your editor to automatically run these linters for you (either on save, or some other keyboard shortcut). However, this is not required. The linters can be run from the command line as well.

To run the linter, execute the following:

```

yarn lint

```

This should output something like the following:

```

yarn run v1.16.0

$ yarn lint:prettier && yarn lint:tslint:src && yarn lint:tslint:tests

$ prettier --write './{src,tests}/\*\*/\*.{ts,js,json}' --loglevel error

$ tslint --project tsconfig.json --fix

$ tslint --project tests/tsconfig.json --fix

? Done in 7.89s.

```

The linter is very strict about the format and style of your code. Anytime it finds something that doesn't comply with the project's formatting and style standards, it will try to automatically rewrite the file to comply. Note that not all linter errors can be fixed automatically. You will have to manually fix those that can't.

## Running the Type Checker

`pdf-lib` is written in TypeScript. This means that the project's source code is contained in `.ts` files. All source code must be correctly typed in order for the tests to run, and for the project to compile.

To run the type checker, execute the following:

```

yarn typecheck

```

This should output something like the following:

```

yarn run v1.16.0

$ tsc --noEmit

? Done in 1.38s.

```

This means your code is correctly typed. If the command fails, then your code has incorrect or missing types somewhere.

## Debugging Tips

Oftentimes when dealing with object offsets and cross references tables/streams, you'll have to deal with byte offsets within a file. The following command can be used (on Linux and Mac machines) to view a given number of bytes at a particular offset:

```bash

cat foo.pdf | tail -c +OFFSET | head -c NUM\_BYTES

```

For example, to view the first 100 bytes following the offset 560477 (aka the byte range 560477-560577), you can run:

```bash

cat foo.pdf | tail -c +560477 | head -c 100

```