

# Node and Typescript

- [React Native on Android](#)
- [Joplin Development](#)
- [Knex](#)
- [NPM and Gitlab](#)
- [Publishing Type Definitions](#)
- [AMQPLib and RabbitMQ](#)
- [Debugging Complex TS Projects in VSCode](#)
- [Handling File Uploads in Express](#)
- [Logging and Winston](#)
- [Fetch](#)
- [npmrc](#)
- [Node Version Management](#)

# React Native on Android

## Installing Dev Environment

On ubuntu you can install the android SDK with `apt-get install android-sdk` .

You'll need a target device to test your app against. You can plug it in and use adb to do the hard work. Make sure to install [the Expo App](#) on your target device - this will serve as a host on the device that allows it to run your new app in development and do hot-reloading of your code.

## Creating a Project

Using npx [create a new project with the tooling provided by React Native](#) like so:

```
npx create-expo-app AwesomeProject
cd AwesomeProject
npm start # you can also use: npx expo start
```

## Running the Project

Start the app with `npm run android` you might get error messages about not knowing where android home is. On ubuntu android home installed from deb packages is normally at `/usr/lib/android-sdk` ([source](#)) so just run:

```
export ANDROID_HOME=/usr/lib/android-sdk
```

 before you try to launch the app.

---

## Theming

I'm looking at using [Native Base](#) as a component library in my apps.

## Persisting Data

React Native is pretty modular so you need to use libraries to do most things. Use [a storage library](#) to store data. This [MMKV storage library](#) appears to be pretty efficient and allows you to store state across multiple databases very easily and quickly

## Navigation

Using [react-navigation](#) to manage navigation within the app and show different views. The [hello world tutorial](#) shows how this works

## Using the Camera

The [expo-camera](#) library seems to be a nice way to do cross-device camera stuff. I got it working very quickly.

### Working Minimal Example Camera App

```
import { StatusBar } from 'expo-status-bar';
import { Camera, CameraType } from 'expo-camera';
import { useState } from 'react';
import { Button, StyleSheet, Text, TouchableOpacity, View } from 'react-native';

export default function App() {
  const [type, setType] = useState(CameraType.back);
  const [permission, requestPermission] = Camera.useCameraPermissions();

  if (!permission) {
    return (<View><Text>No camera m8</Text></View>)
  }

  if (!permission.granted) {
    return(<View>
      <Text>Clicky to grant perms m8</Text>
      <Button title='Grant Perms' onPress={requestPermission}/>
    </View>)
  }

  function toggleCameraType() {
```

```

    setType(current => (current === CameraType.back ? CameraType.front : CameraType.back));
  }

  return (
    <View style={styles.container}>
      <Camera style={styles.camera} type={type}>
        <View style={styles.buttonContainer}>
          <TouchableOpacity style={styles.button} onPress={toggleCameraType}>
            <Text style={styles.text}>Flip Camera</Text>
          </TouchableOpacity>
        </View>
      </Camera>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
  },
  camera:{
    flex: 1,
  },
  button: {
    flex: 1,
    alignSelf: 'flex-end',
    alignItems: 'center',
  },
  buttonContainer: {
    flex: 1,
    flexDirection: 'row',
    backgroundColor: 'transparent',
    margin: 64,
  },
  text:{

  }
});

```

###

# Joplin Development

[Joplin](#) features a rich plugin architecture which allows third party developers to build plugins to enable non-standard behaviours.

Joplin is written in Javascript/Typescript and built on top of electron. Plugins must therefore run in an electron runtime.

## Finding Global Settings Values

There are some use cases that might require you to know what settings the user has entered for areas of the application outside of your plugin (for example you might need to know if the user has sync enabled so that you can avoid destructive operations while synchronisation is happening).

The solution is to use the [joplin.settings.globalValue](#) api function. The global settings keys are not documented and could change. The dev team suggest that developers look for useful values in [Settings.ts](#) in the main project repo.

## Publishing Plugins

The official Joplin plugin marketplace/selector retrieves items from [this github repo](#) which in turns checks npm for packages that are published with the keyword `joplin-plugin`. So all you need to do is ensure that your project follows the standard structure of a joplin plugin (e.g. comes with a manifest.json, has the corresponding npm steps) and has this keyword.

The [yeoman](#) template provided by the joplin team comes pre-configured ready for publication of plugins.

# Knex

- ORM for node.js with Typescript compatibility

# NPM and Gitlab

Gitlab has a built in package repository that can be used as a stand in for NPM's global repo.

Best practice is to map a [scope](#) to your repository in your `.npmrc` file and in your package's `package.json` file.

Gitlab uses CI tokens to authenticate against the npm repository.



# Publishing Type Definitions

It can be useful to be able to publish these types in custom NPM repos e.g. [Gitlab](#)

## Configuring `package.json` File

### Including `src` and `dist` folders

use `files` key in `package.json` to indicate which directories should be published ( <https://stackoverflow.com/questions/67523877/index-d-ts-file-not-published-to-npm>). By default you likely want to publish both the source code in `src` and the compiled/transpiled typescript->js files in `dist`

e.g.

```
{
  "files": [
    "dist",
    "src"
  ],
}
```

### Including types and main

You also need to include a `types` property which tells npm where your defined types (`index.d.ts`) sits - this is likely to be in `dist/index.d.ts` if you follow the conventions in this document.

### Main module

In a node module you'd probably use `src/index.js` but we want to use `dist/index.js` - this will be the compiled version of our typescript module.

# Configuring `tsconfig.json`

You probably want something like this:

```
{
  "compilerOptions": {
    "sourceMap": true,
    "outDir": "dist",
    "strict": true,
    "lib": [
      "esnext"
    ],
    "allowJs": true,
    "declaration": true,
    "esModuleInterop": true
  },

  "include": ["src/**/*"],
}
```

Use `dist` for our `outdir` so that the output from transpilation goes to the right place. Use `declaration: true` to have `tsc` produce an `index.d.ts` declaration file and plonk it into the `dist` folder so that it is picked up by the `package.json` we configured above.

# AMQP Lib and RabbitMQ

## Channel closed by server: 406 (PRECONDITION-FAILED) with message "PRECONDITION\_FAILED"

As explained by [this article](#) this implies that your channel has consumed a message without ACKing or NACKing it and it has timed out. Make sure to ACK or NACK all messages when you receive them - typically after processing just in case something goes wrong there.

```
this.channel?.consume(q?.queue, (msg: amqplib.ConsumeMessage | null) => {  
  
  if(!msg){  
    throw new Error(`Received a null message - weirdness.`)  
  }  
  
  try{  
  
    const result = callback(msg)  
  
    this.channel?.ack(msg)  
  
    return result  
  
  }catch(err) {  
    console.log(err)  
    this.channel?.nack(msg)  
  }  
  
})
```

# Debugging Complex TS Projects in VSCode

## launch.json Config File

This page assumes a build config based on the one in [Publishing Type Definitions](#) with a `src` and a `dist` folder.

We can use `npm link` to link `project1` to `project2` and then we can add the dist folders from both projects to `outFiles` so that they are picked up by the vscode debugger.

We also want to make sure that `sourceMaps` is set to true so that the debugger picks up the source maps in the dist folders and maps them onto the corresponding `src` folders.

```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "program": "${workspaceFolder}/project1/dist/index.js",
      "preLaunchTask": "Build All",
      "outFiles": [
        "${workspaceFolder}/project1/dist/**/*.js",
        "${workspaceFolder}/project2/dist/**/*.js"
      ],
      "sourceMaps": true,
      "sourceMapRenames": true
    }
  ]
}
```

The launch definition can fire off a pre-launch task so we can use that to always ensure that the transpiled js code is up to date before we launch it. The below configuration defines the builds and dependencies between them

## tasks.json Configuration

This file contains build tasks that the launch.json file can depend on before launching.

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "typescript",
      "tsconfig": "projecta/tsconfig.json",
      "problemMatcher": [
        "$tsc"
      ],
      "group": "build",
      "label": "tsc: build - projecta/tsconfig.json",
      "dependsOn": ["tsc: build - projectb/tsconfig.json"]
    },
    {
      "type": "typescript",
      "tsconfig": "projectb/tsconfig.json",
      "problemMatcher": [
        "$tsc"
      ],
      "group": "build",
      "label": "tsc: build - projectb/tsconfig.json"
    },
    {
      "label": "Build All",
      "dependsOn": [
        "tsc: build - projecta/tsconfig.json",
        "tsc: build - projectb/tsconfig.json"
      ]
    }
  ]
}
```

We can define two typescript build operations - linking them together with dependencies and we can also define a `Build All` job which is just a dummy job that doesn't do anything but requires the ts builds to complete before it can be considered complete itself.

# Handling File Uploads in Express

## Express and Multi-Part Uploads

Historically I've used the express [Multer](#) middleware and associated typescript types ( `@types/multer` ).

Multer accepts files encoded as part of a `multipart/form-data` request. The file is added to the `req` object as `req.file : Express.Multer.File`

This is done by adding a middleware to the route that accepts files like so:

```
import multer from 'multer'

...

const uploadMiddleware = multer({limits:{
  fileSize: config.uploads.maxSize
}})

router.post('/blah',
  uploadMiddleware.single('file'),
  controllerFunction );
```

## Storing Uploaded Files in S3

It is possible to use minio to immediately pass the file through to an S3 layer via the buffer object. Using a Minio client object:

```
async function handleFileUpload(req: Express.Request, res: Express.Response) {  
  
  const fileObj : Express.Multer.File = req.file  
  let file = await Minio.putObject(config.minio.bucket, objName, fileObj.buffer);  
  
  return {"status":"uploaded", file}  
  
}
```

## Dealing with CSVs

Use [neat-csv](#) for a promisy experience



# Logging and Winston

Winston is a fancy logging library for node.

## Using Common Loggers Between Packages

As per [this stackoverflow](#) post ([mirror](#)):

Declare and export your winston logger object and from different locations within your app.

# Fetch

The Fetch API was [adopted as a standard replacement for XMLHttpRequest](#) in the last few years. It is usually available by default in browser code. However it is not offered out-of-the-box by Node.js

## Cross-Fetch

The [cross-fetch](#) library provides cross-platform fetch API compatibility for node-like environments.

It can be installed with `npm i --save cross-fetch`.

It can be used as an ES6 module via

```
import fetch from 'cross-fetch';
```

# npmrc

npmrc files registry namespaces and credentials that allow users to authenticate against private (and public) npm registries.

NPM looks for a `.npmrc` file in the current directory and prioritises this over the one in your user `$HOME` directory.

# Node Version Management

[Volta](#) is a modern NodeJS runtime and tooling manager similar to [NVM](#). An advantage of Volta over nvm is that it can store metadata about the running version of node in a `package.json` file and install the correct version of node.

Install specific tools and runtimes with `volta install node@<version> npm@<version>`

## Example Package.json Content

When you run `volta pin node@18.19.0` you might find that something similar to the below is injected into your package.json file:

```
"volta": {  
  "node": "18.19.0"  
}
```