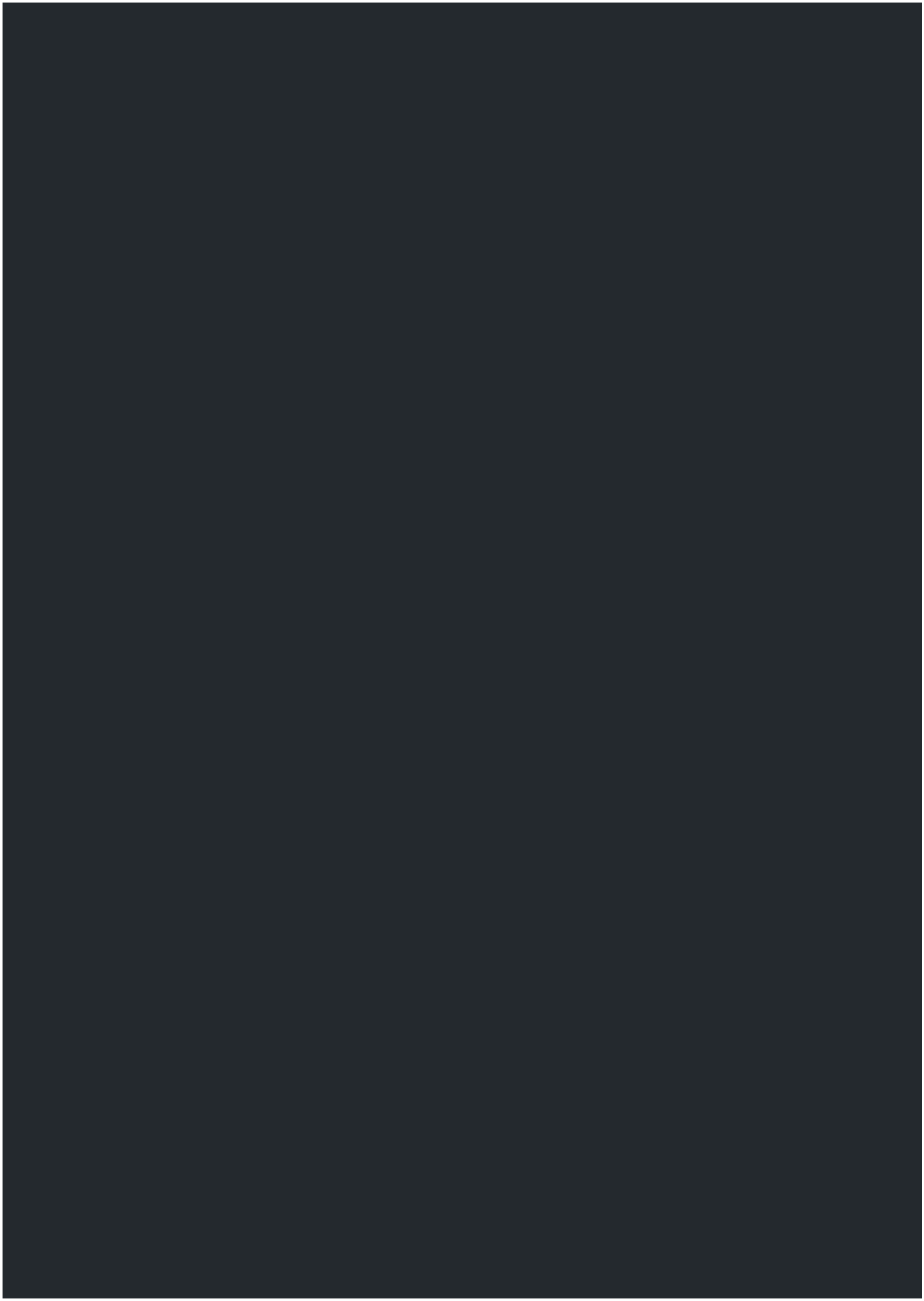
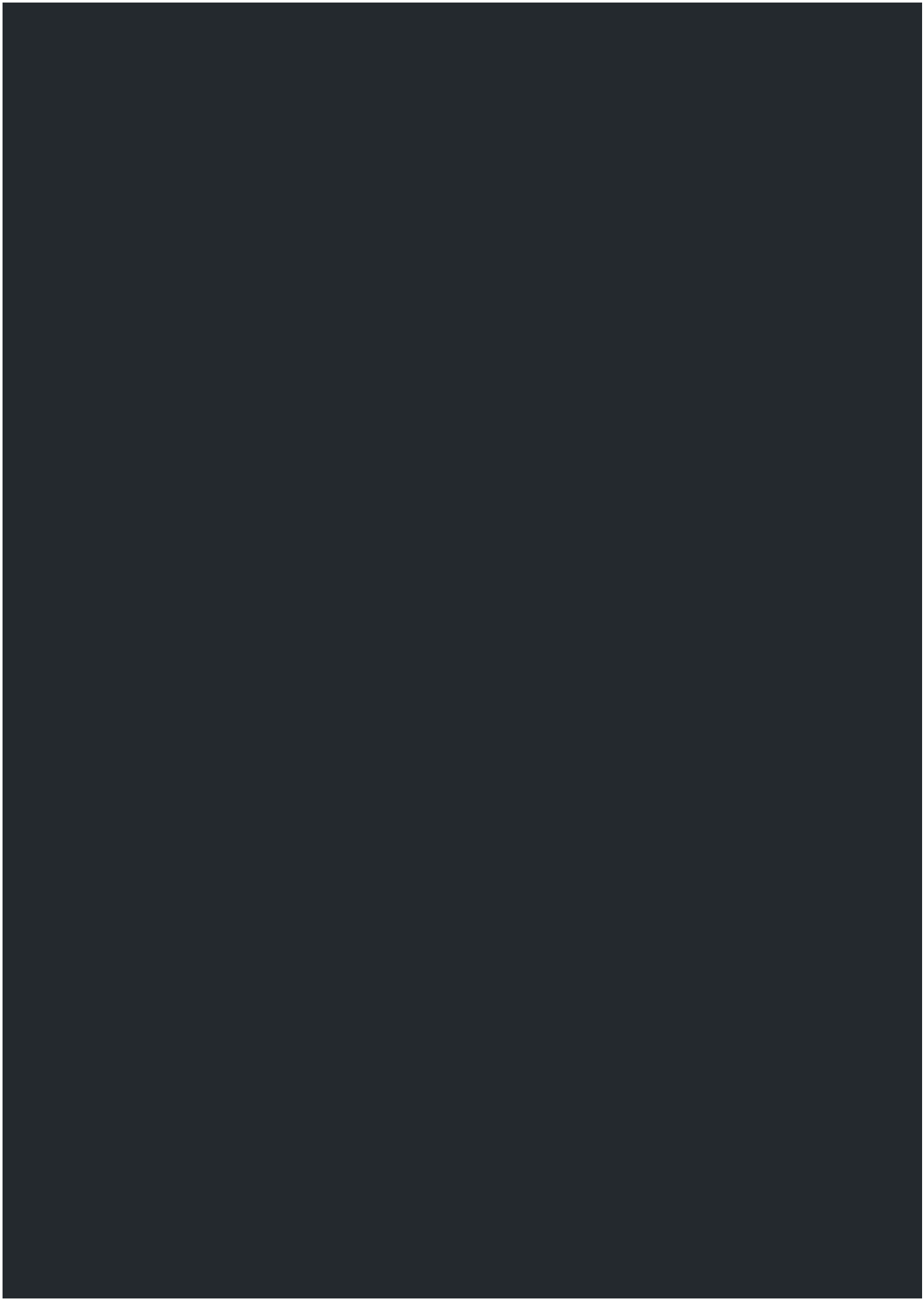
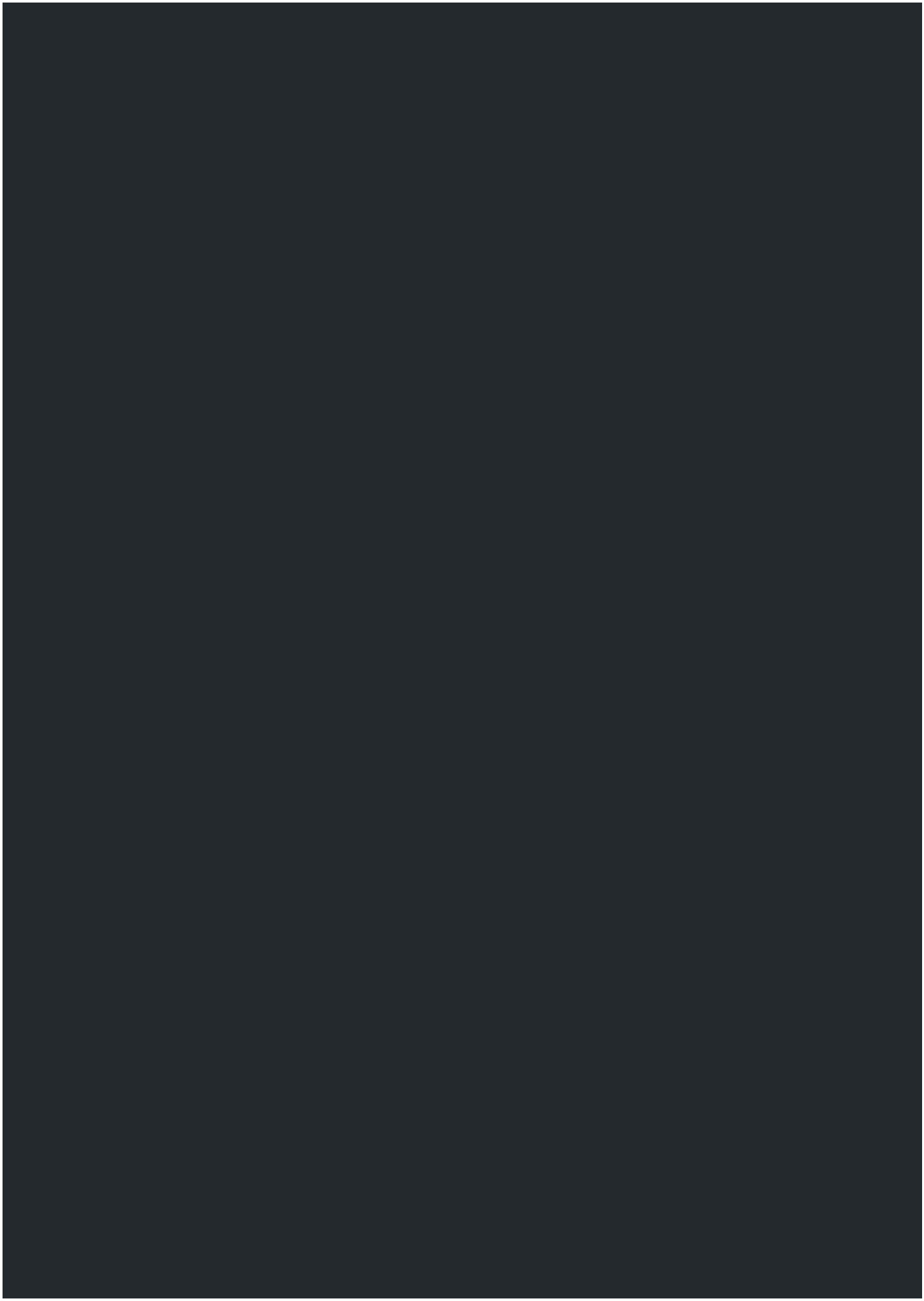
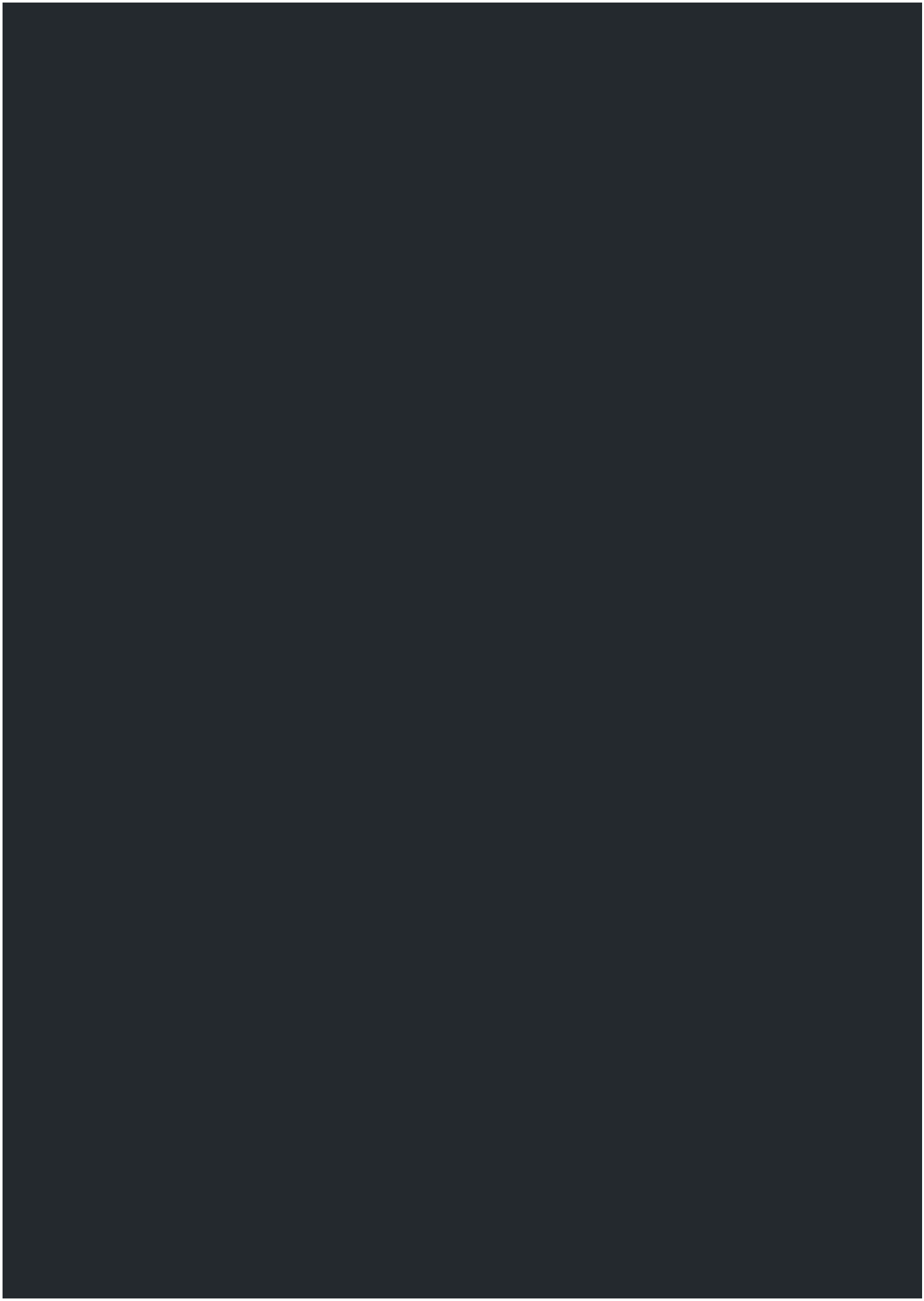
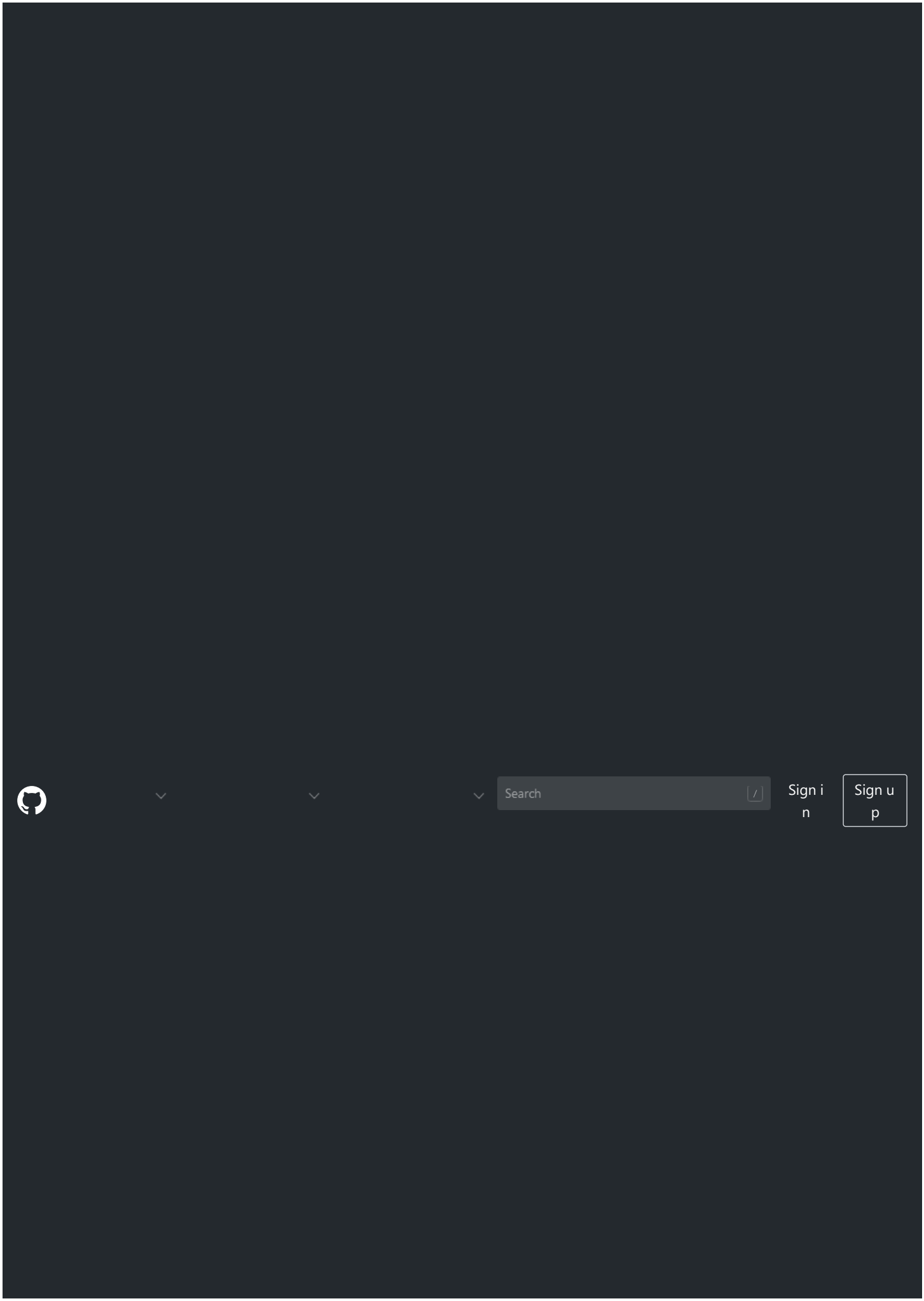
Enterprise Marketplace

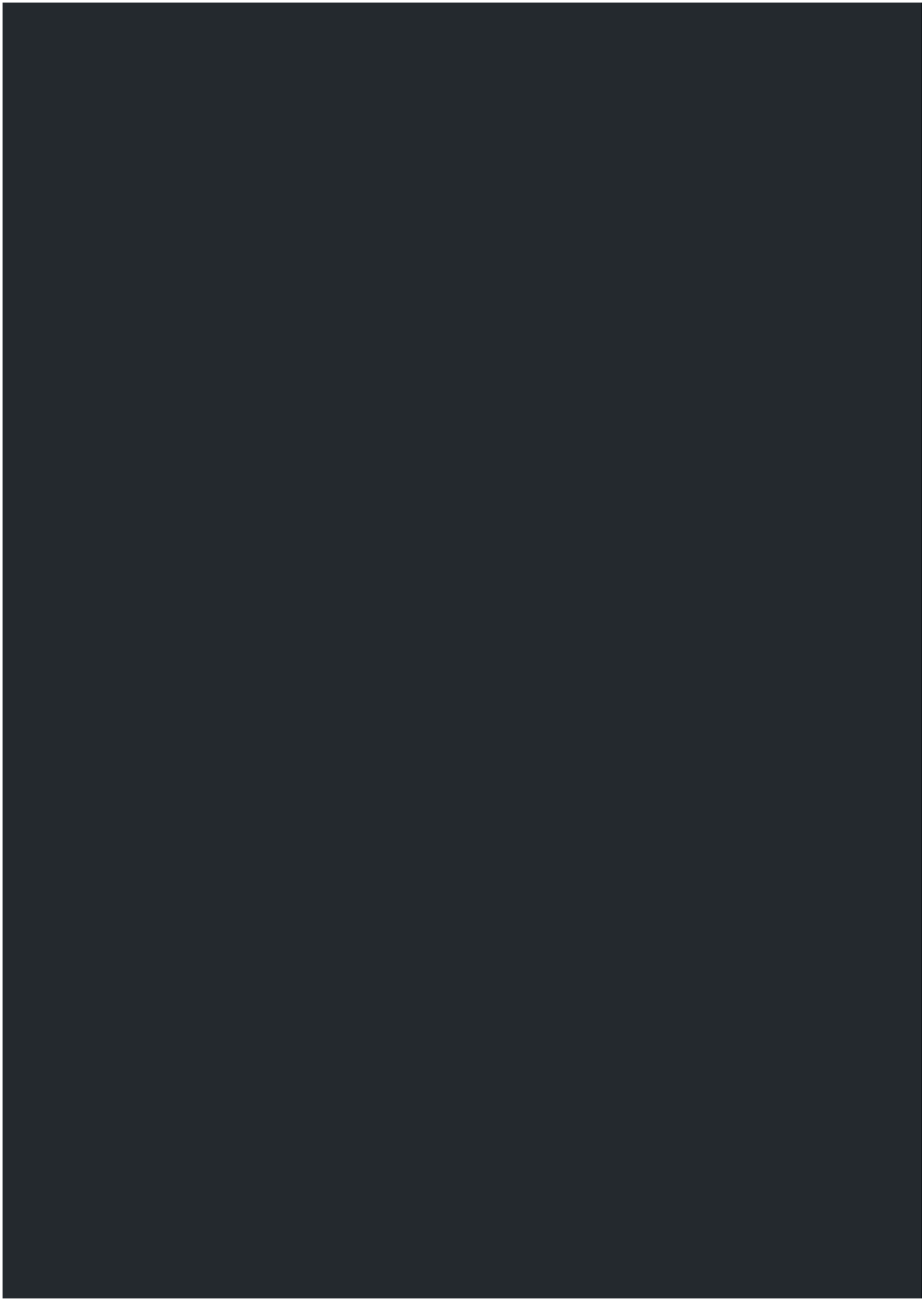


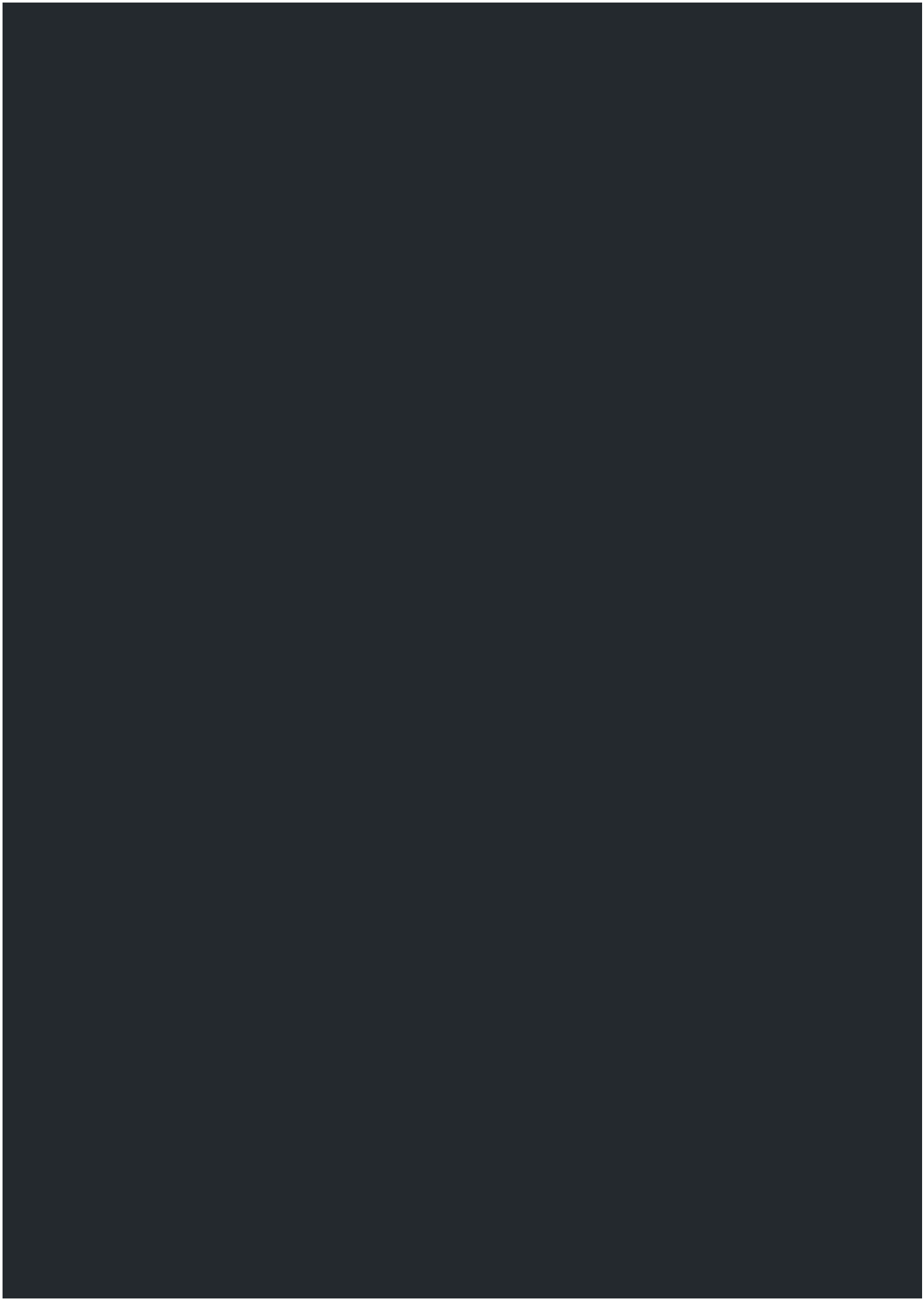


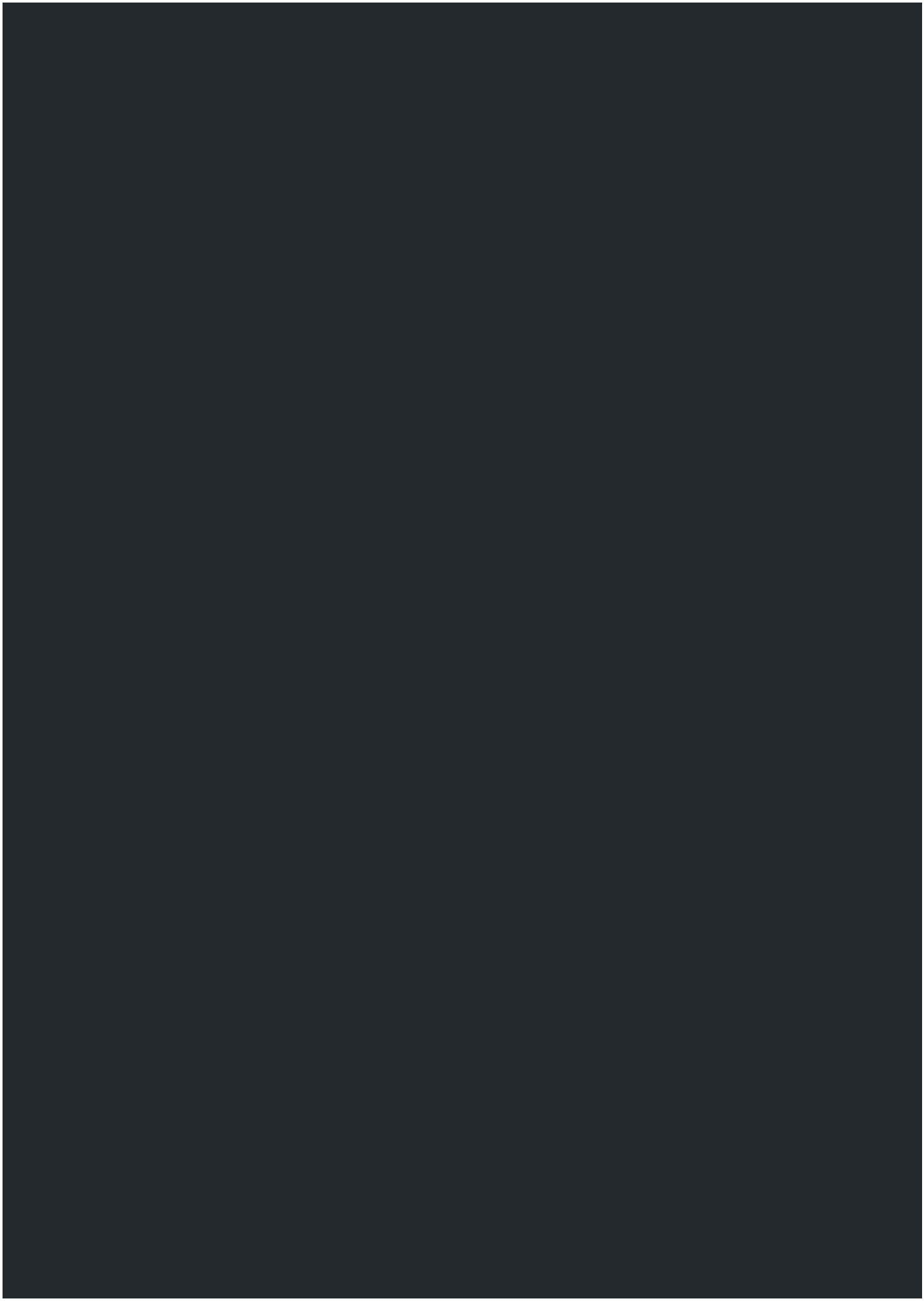


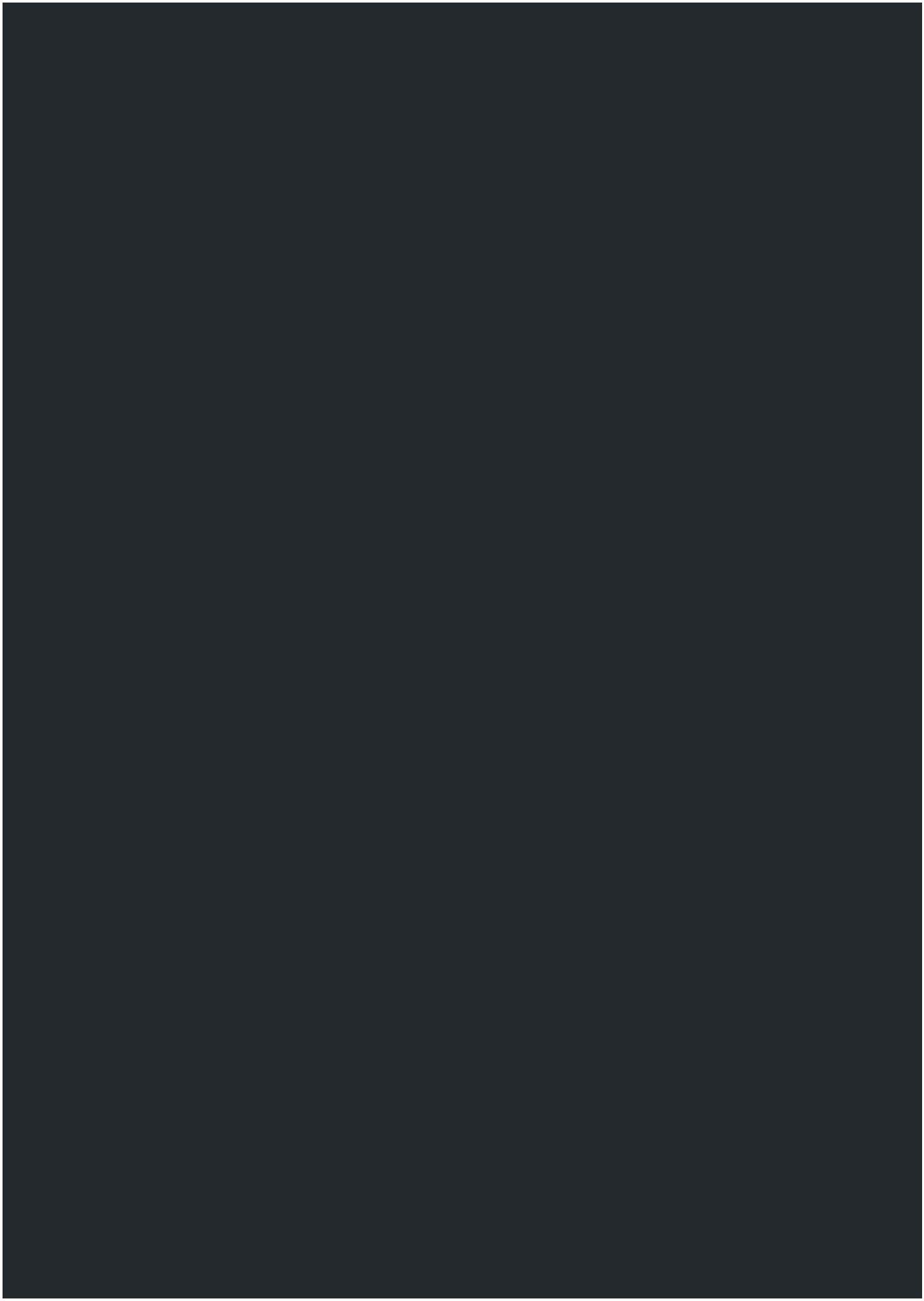


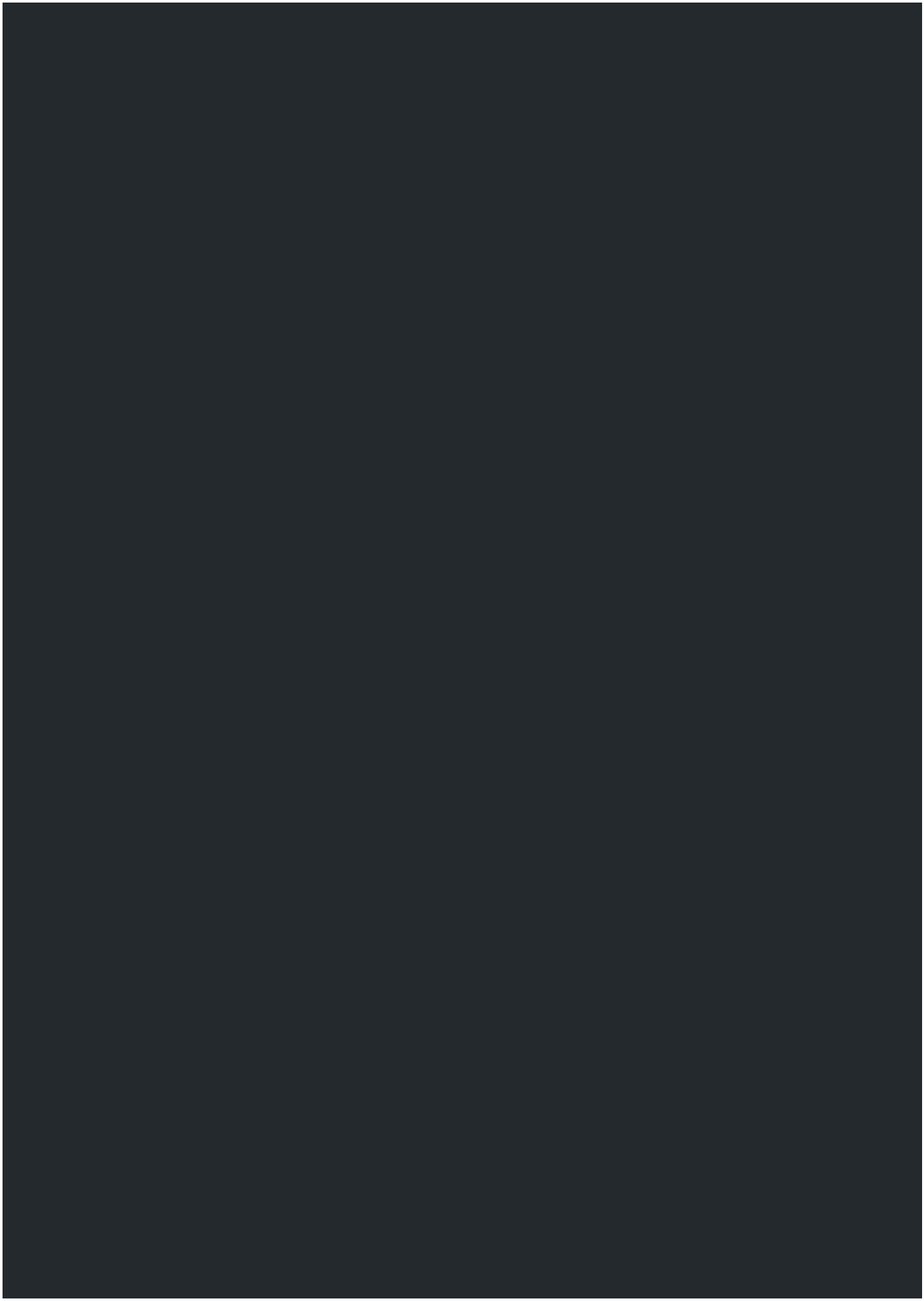

Search
/
Sign in
Sign up











Cheatsheets for experienced React developers getting started with TypeScript https://twitter.com/swyx/status/10027...

240 commits 3 branches 0 releases 32 contributors MIT

Branch: master New pull request Find File Clone or download

sw-yx use "reverse" defaultProp type pattern ... Latest commit 6cbdb4d a day ago

.github/ISSUE_TEMPLATE	add issue templates	2 months ago
ADVANCED.md	add warning notice for tslint	2 days ago
CONTRIBUTING.md	add MIGRATING	2 months ago
HOC.md	Update HOC.md	a month ago
LICENSE	Initial commit	10 months ago
MIGRATING.md	Update MIGRATING.md	a month ago
README.md	use "reverse" defaultProp type pattern	a day ago

README.md

React+TypeScript Cheatsheets

Cheatsheets for experienced React developers getting started with TypeScript

Basic | Advanced | Migrating | HOC | 中文翻译 | Contribute! | Ask!

👋 This repo is maintained by @swyx, @ferdaber and @IslamAttrash, we're so happy you want to try out TypeScript with React!
If you see anything wrong or missing, please file an issue! 👍

All React + TypeScript Cheatsheets

- The Basic Cheatsheet (`/README.md`) is focused on helping React devs just start using TS in React apps
  - focus on opinionated best practices, copy+pastable examples
  - explains some basic TS types usage and setup along the way
  - answers the most Frequently Asked Questions
  - does not cover generic type logic in detail. Instead we prefer to teach simple troubleshooting techniques for newbies.
  - The goal is to get effective with TS without learning *too much* TS.
- The Advanced Cheatsheet (`/ADVANCED.md`) helps show and explain advanced usage of generic types for people writing reusable type utilities/functions/render prop/higher order components and TS+React libraries.
  - It also has miscellaneous tips and tricks for pro users.
  - Advice for contributing to DefinitelyTyped
  - The goal is to take *full advantage* of TypeScript.
- The Migrating Cheatsheet (`/MIGRATING.md`) helps collate advice for incrementally migrating large codebases from JS or Flow, from people who have done it.
  - We do not try to convince people to switch, only to help people who have already decided
  - ⚠️This is a new cheatsheet, all assistance is welcome
- The HOC Cheatsheet (`/HOC.md`) specifically teaches people to write HOCs with examples.
  - Familiarity with Generics is necessary.
  - ⚠️This is the newest cheatsheet, all assistance is welcome

Basic Cheatsheet Table of Contents

Section 1: Setup

Prerequisites

1. good understanding of React
2. familiarity with TypeScript Types (2ality's guide is helpful)
3. having read the TypeScript section in the official React docs.

This guide will always assume you are starting with the latest TypeScript version. Notes for older versions will be in expandable `<details>` tags.

React + TypeScript Starter Kits

1. Create React App v2.1+ with Typescript: `npx create-react-app my-new-react-typescript-app --typescript`
   - We used to recommend `create-react-app-typescript` but it is now deprecated. see migration instructions
2. Basarat's guide for manual setup of React + TypeScript + Webpack + Babel
   - In particular, make sure that you have `@types/react` and `@types/react-dom` installed (Read more about the DefinitelyTyped project if you are unfamiliar)
   - There are also many React + TypeScript boilerplates, please see our Resources list below.

Import React

```
import * as React from 'react'; import * as ReactDOM from 'react-dom';
```

In TypeScript 2.7+, you can run TypeScript with `--allowSyntheticDefaultImports` (or add `"allowSyntheticDefaultImports": true` to tsconfig) to import like in regular jsx:

```
import React from 'react'; import ReactDOM from 'react-dom';
```

▶ Explanation

Section 2: Getting Started

Function Components

You can specify the type of props as you use them and rely on type inference:

```
const App = ({ message }: { message: string }) => <div>{message}</div>;
```

Or you can use the provided generic type for function components:

```
const App: React.FC<{ message: string }> = ({ message }) => ( <div>{message}</div> ); //
React.FunctionComponent also works
```

▶ What's the difference?

▶ Minor Pitfalls

Hooks

Hooks are supported in `@types/react` from v16.8 up.

useState

Type inference works very well most of the time:

```
const [val, toggle] = useState(false); // `val` is inferred to be a boolean, `toggle` only takes booleans
```

See also the Using Inferred Types section if you need to use a complex type that you've relied on inference for.

However, many hooks are initialized with null-ish default values, and you may wonder how to provide types. Explicitly declare the type, and use a union type:

```
const [user, setUser] = useState<IUser | null>(null); // later... setUser(newUser);
```

useRef

When using `useRef`, you have two options when creating a ref container that does not have an initial value:

```
const ref1 = useRef<HTMLElement>(null) const ref2 = useRef<HTMLElement | null>(null)
```

The first option will make `ref1.current` read-only, and is intended to be passed in to built-in `ref` attributes that React will manage (because React handles setting the `current` value for you).

The second option will make `ref2.current` mutable, and is intended for "instance variables" that you manage yourself.

useEffect

When using `useEffect`, take care not to return anything other than a function or `undefined`, otherwise both TypeScript and React will yell at you. This can be subtle when using arrow functions:

```
function DelayedEffect(props: { timerMs: number }) { const { timerMs } = props; // bad! setTimeout implicitly returns a number because the arrow function body isn't wrapped in curly braces useEffect(() => setTimeout(() => {/* do stuff */}, timerMs), [timerMs]) return null }
```

Custom Hooks

If you are returning an array in your Custom Hook, you will want to avoid type inference as Typescript will infer a union type (when you actually want different types in each position of the array). Instead, assert or define the function return types:

```
export function useLoading() { const [isLoading, setState] = React.useState(false); const load = (aPromise: Promise<any>) => { setState(true); return aPromise.finally(() => setState(false)); }; return [isLoading, load] as [ boolean, (aPromise: Promise<any>) => Promise<any> ]; }
```

A helper function that automatically types tuples can also be helpful if you write a lot of custom hooks:

```
function tuplify<T extends any[]>(...elements: T) { return elements } function useArray() { const numberValue = useRef(3).current const functionValue = useRef(() => {}).current return [numberValue, functionValue] // type is (number | (() => void))[] } function useTuple() { const numberValue = useRef(3).current const functionValue = useRef(() => {}).current return tuplify(numberValue, functionValue) // type is [number, () => void] }
```

The React team recommends that custom hooks that return more than two values should use proper objects instead of tuples, however.

If you are writing a React Hooks library, don't forget that you should also expose your types for users to use.

Example React Hooks + TypeScript Libraries:

- https://github.com/mweststrate/use-st8
- https://github.com/palmerhq/the-platform
- https://github.com/sw-yx/hooks

Something to add? File an issue.

Class Components

Within TypeScript, `React.Component` is a generic type (aka `React.Component<PropType, StateType>`), so you want to provide it with (optional) prop and state type parameters:

```
type MyProps = { // using `interface` is also ok message: string; }; type MyState = { count: number; // like this }; class App extends React.Component<MyProps, MyState> { state: MyState = { // optional second annotation for better type inference count: 0 }; render() { return ( <div> {this.props.message} {this.state.count} </div> ); } }
```

Don't forget that you can export/import/extend these types/interfaces for reuse.

▶ Why annotate `state` twice?

▶ No need for `readonly`

Class Methods: Do it like normal, but just remember any arguments for your functions also need to be typed:

```
class App extends React.Component<{ message: string }, { count: number }> { state = { count: 0 }; render() { return ( <div onClick={() => this.increment(1)}> {this.props.message} {this.state.count} </div> ); } increment = (amt: number) => { // like this this.setState(state => ({ count: state.count + amt })); }; }
```

Class Properties: If you need to declare class properties for later use, just declare it like `state`, but without assignment:

```
class App extends React.Component<{ message: string; }> { pointer: number; // like this componentDidMount() { this.pointer = 3; } render() { return ( <div> {this.props.message} and {this.pointer} </div> ); } }
```

Something to add? File an issue.

Typing defaultProps

For Typescript 3.0+, type inference should work, although some edge cases are still problematic. Just type your props like normal, except don't use `React.FC`.

```
// ///////////////// // function components // ///////////////// type Props = { age: number } & typeof defaultProps; const defaultProps = { who: 'Johny Five', }; const Greet = (props: Props) => { /*...*/ }; Greet.defaultProps = defaultProps
```

For Class components, there are a couple ways to do it(including using the `Pick` utility type) but the recommendation is to "reverse" the props definition:

```
type GreetProps = typeof Greet.defaultProps & { age: number } class Greet extends React.Component<GreetProps> { static defaultProps = { name: 'world' } /*...*/ } // Type-checks! No type assertions needed! let el = <Greet age={3} />;
```

▶ Why does React.FC break defaultProps?

▶ Typescript 2.9 and earlier

Something to add? File an issue.

Types or Interfaces?

`interface`s are different from `type`s in TypeScript, but they can be used for very similar things as far as common React uses cases are concerned. Here's a helpful rule of thumb:

- always use `interface` for public API's definition when authoring a library or 3rd party ambient type definitions.
- consider using `type` for your React Component Props and State, because it is more constrained.

Types are useful for union types (e.g. `type MyType = TypeA | TypeB`) whereas Interfaces are better for declaring dictionary shapes and then `implementing` or `extending` them.

▶ Useful table for Types vs Interfaces

Something to add? File an issue.

Basic Prop Types Examples

```
type AppProps = { message: string; count: number; disabled: boolean; /** array of a type! */ names: string[]; /** string literals to specify exact string values, with a union type to join them together */ status: 'waiting' | 'success'; /** any object as long as you dont use its properties (not common) */ obj: object; obj2: {}; // same /** an object with defined properties (preferred) */ obj3: { id: string; title: string; }; /** array of objects! (common) */ objArr: { id: string; title: string; }[]; /** any function as long as you don't invoke it (not recommended) */ onSomething: Function; /** function that doesn't take or return anything (VERY COMMON) */ onClick: () => void; /** function with named prop (VERY COMMON) */ onChange: (id: number) => void; /** an optional prop (VERY COMMON!) */ optional?: OptionalType; };
```

Notice we have used the TSDoc `/** comment */` style here on each prop. You can and are encouraged to leave descriptive comments on reusable components. For a fuller example and discussion, see our Commenting Components section in the Advanced Cheatsheet.

Useful React Prop Type Examples

```
export declare interface AppProps { children1: JSX.Element; // bad, doesnt account for arrays children2: JSX.Element | JSX.Element[]; // meh, doesnt accept functions children3: React.ReactChildren; // despite the name, not at all an appropriate type; it is a utility children3: React.ReactChild[]; // better
```

```
children: React.ReactNode; // best, accepts everything functionChildren: (name: string) =>
React.ReactNode; // recommended function as a child render prop type style?: React.CSSProperties; // to
pass through style props onChange?: React.FormEventHandler<HTMLInputElement>; // form events! the generic
parameter is the type of event.target props: Props &
React.PropsWithoutRef<JSX.IntrinsicElements['button']>; // to impersonate all the props of a button
element without its ref }
```

▶ JSX.Element vs React.ReactNode?

Something to add? File an issue.

Forms and Events

If performance is not an issue, inlining handlers is easiest as you can just use type inference:

```
const el = ( <button onClick={event => { /* ... */ }} /> );
```

But if you need to define your event handler separately, IDE tooling really comes in handy here, as the @type definitions come with a wealth of typing. Type what you are looking for and usually the autocomplete will help you out. Here is what it looks like for an `onChange` for a form event:

```
class App extends React.Component< {}, { // no props text: string; } > { state = { text: '' }; // typing
on RIGHT hand side of = onChange = (e: React.FormEvent<HTMLInputElement>): void => { this.setState({ text:
e.currentTarget.value }); }; render() { return ( <div> <input type="text" value={this.state.text}
onChange={this.onChange} /> </div> ); } }
```

Instead of typing the arguments and return values with `React.FormEvent<>` and `void`, you may alternatively apply types to the event handler itself (*contributed by @TomasHubelbauer*):

```
// typing on LEFT hand side of = onChange: React.ChangeEventHandler<HTMLInputElement> = (e) => {
this.setState({text: e.currentTarget.value}) }
```

▶ Why two ways to do the same thing?

Context

Contributed by: @jpavon

Using the new context API `React.createContext`:

```
interface ProviderState { themeColor: string; } interface UpdateStateArg { key: keyof ProviderState;
value: string; } interface ProviderStore { state: ProviderState; update: (arg: UpdateStateArg) => void; }
const Context = React.createContext({} as ProviderStore); // type assertion on empty object class Provider
extends React.Component<{}, ProviderState> { public readonly state = { themeColor: 'red' }; private update
= ({ key, value }: UpdateStateArg) => { this.setState({ [key]: value }); }; public render() { const store:
ProviderStore = { state: this.state, update: this.update }; return ( <Context.Provider value={store}>
{this.props.children}</Context.Provider> ); } } const Consumer = Context.Consumer;
```

Something to add? File an issue.

forwardRef/createRef

`createRef`:

```
class CssThemeProvider extends React.PureComponent<Props> { private rootRef =
React.createRef<HTMLDivElement>(); // like this render() { return <div ref={this.rootRef}>
{this.props.children}</div>; } }
```

`forwardRef`:

```
type Props = { children: React.ReactNode; type: 'submit' | 'button' }; export type Ref =
HTMLButtonElement; export const FancyButton = React.forwardRef<Ref, Props>((props, ref) => ( <button ref=
{ref} className="MyClassName" type={props.type}> {props.children} </button> ));
```

If you are grabbing the props of a component that forwards refs, use `ComponentPropsWithRef`.

More info: https://medium.com/@martin_hotell/react-refs-with-typescript-a32d56c4d315

Something to add? File an issue.

Portals

Using `ReactDOM.createPortal`:

```
const modalRoot = document.getElementById('modal-root') as HTMLElement; // assuming in your html file has
a div with id 'modal-root'; export class Modal extends React.Component { el: HTMLElement =
document.createElement('div'); componentDidMount() { modalRoot.appendChild(this.el); }
componentWillUnmount() { modalRoot.removeChild(this.el); } render() { return
ReactDOM.createPortal(this.props.children, this.el); } }
```

▶ Context of Example

Error Boundaries

Not written yet.

Something to add? File an issue.

Concurrent React/React Suspense

Not written yet. watch https://github.com/sw-yx/fresh-async-react for more on React Suspense and Time Slicing.

Something to add? File an issue.

Basic Troubleshooting Handbook: Types

Facing weird type errors? You aren't alone. This is the hardest part of using TypeScript with React. Be patient - you are learning a new language after all. However, the more you get good at this, the less time you'll be working *against* the compiler and the more the compiler will be working *for* you!

Try to avoid typing with `any` as much as possible to experience the full benefits of typescript. Instead, let's try to be familiar with some of the common strategies to solve these issues.

Union Types and Type Guarding

Union types are handy for solving some of these typing problems:

```
class App extends React.Component< {}, { count: number | null; // like this } > { state = { count: null };
render() { return <div onClick={() => this.increment(1)}>{this.state.count}</div>; } increment = (amt:
number) => { this.setState(state => ({ count: (state.count || 0) + amt })); }; }
```

Type Guarding: Sometimes Union Types solve a problem in one area but create another downstream. Learn how to write checks, guards, and assertions (also see the Conditional Rendering section below). For example:

```
interface Admin { role: string: } interface User { email: string; } // Method 1: use `in` keyword function
redirect(usr: Admin | User) { if("role" in usr) { // use the `in` operator for typeguards since TS 2.7+
routeToAdminPage(usr.role); } else { routeToHomePage(usr.email); } } // Method 2: custom type guard, does
the same thing in older TS versions or where `in` isnt enough function isAdmin(usr: Admin | User): usr is
Admin { return (<Admin>usr).role !==undefined }
```

If you need `if...else` chains or the `switch` statement instead, it should "just work", but look up Discriminated Unions if you need help. (See also: Basarat's writeup). This is handy in typing reducers for `useReducer` or Redux.

Optional Types

If a component has an optional prop, add a question mark and assign during destructure (or use defaultProps).

```
class MyComponent extends React.Component<{ message?: string; // like this }> { render() { const { message
= 'default' } = this.props; return <div>{message}</div>; } }
```

You can also use a `!` character to assert that something is not undefined, but this is not encouraged.

Something to add? File an issue with your suggestions!

Enum Types

Enums in TypeScript default to numbers. You will usually want to use them as strings instead:

```
export enum ButtonSizes { default = 'default', small = 'small', large = 'large' }
```

Usage:

```
export const PrimaryButton = ( props: Props & React.HTMLProps<HTMLButtonElement> ) => <Button size=
{ButtonSizes.default} {...props} />;
```

A simpler alternative to enum is just declaring a bunch of strings with union:

```
export declare type Position = 'left' | 'right' | 'top' | 'bottom';
```

▶ Brief Explanation

Type Assertion

Sometimes TypeScript is just getting your type wrong, or union types need to be asserted to a more specific type to work with other APIs, so assert with the `as` keyword. This tells the compiler you know better than it does.

```
class MyComponent extends React.Component<{ message: string; }> { render() { const { message } =
this.props; return ( <Component2 message={message as SpecialMessageType}>{message}</Component2> ); } }
```

▶ Explanation

Intersection Types

Adding two types together can be handy, for example when your component is supposed to mirror the props of a native component like a `button`:

```
export interface Props { label: string; } export const PrimaryButton = ( props: Props &
React.HTMLProps<HTMLButtonElement> // adding my Props together with the @types/react button provided props
) => <Button {...props} />;
```

Using Inferred Types

Leaning on Typescript's Type Inference is great... until you realize you need a type that was inferred, and have to go back and explicitly declare types/interfaces so you can export them for reuse.

Fortunately, with `typeof`, you won't have to do that. Just use it on any value:

```
const [state, setState] = React.useState({ foo: 1, bar: 2 }); // state's type inferred to be {foo: number, bar: number} const someMethod = (obj: typeof state) => { // grabbing the type of state even though it was inferred // some code using obj setState(obj); // this works };
```

Using Partial Types

Working with slicing state and props is common in React. Again, you don't really have to go and explicitly redefine your types if you use the `Partial` generic type:

```
const [state, setState] = React.useState({ foo: 1, bar: 2 }); // state's type inferred to be {foo: number, bar: number} // NOTE: stale state merging is not actually encouraged in React.useState // we are just demonstrating how to use Partial here const partialStateUpdate = (obj: Partial<typeof state>) => setState({ ...state, ...obj }); // later on... partialStateUpdate({ foo: 2 }); // this works
```

▶ Minor caveats on using `Partial`

The Types I need weren't exported!

This can be annoying but here are ways to grab the types!

- Grabbing the Prop types of a component: Use `React.ComponentProps` and `typeof`, and optionally `Omit` any overlapping types

```
import { Button } from 'library'; // but doesn't export ButtonProps! oh no! type ButtonProps = React.ComponentProps<typeof Button>; // no problem! grab your own! type AlertButtonProps = Omit<ButtonProps, 'onClick'>; // modify const AlertButton: React.FC<AlertButtonProps> = props => ( <Button onClick={() => alert('hello')} {...props} /> );
```

You may also use `ComponentPropsWithoutRef` (instead of ComponentProps) and `ComponentPropsWithRef` (if your component specifically forwards refs)

- Grabbing the return type of a function: use `ReturnType`:

```
// inside some library - return type { baz: number } is inferred but not exported function foo(bar: string) { return { baz: 1 }; } // inside your app, if you need { baz: number } type FooReturn = ReturnType<typeof foo>; // { baz: number }
```

Troubleshooting Handbook: Images and other non-TS/TSX files

Use declaration merging:

```
// declaration.d.ts // anywhere in your project, NOT the same name as any of your .ts/tsx files declare module '*.png' // importing in a tsx file import * as logo from "./logo.png";
```

Related issue: https://github.com/Microsoft/TypeScript-React-Starter/issues/12 and StackOverflow

Troubleshooting Handbook: TSLint

Sometimes TSLint is just getting in the way. Judiciously turning off of things can be helpful. Here are useful tslint disables you may use:

- `/* tslint:disable */` total disable
- `// tslint:disable-line` just this line

- `/* tslint:disable:semicolon */` sometimes prettier adds semicolons and tslint doesn't like it.
- `/* tslint:disable:no-any */` disable tslint restriction on no-any when you WANT to use any
- `/* tslint:disable:max-line-length */` disable line wrapping linting

so on and so forth. there are any number of things you can disable, usually you can look at the error raised in VScode or whatever the tooling and the name of the error will correspond to the rule you should disable.

▶ Explanation

Troubleshooting Handbook: tsconfig.json

You can find all the Compiler options in the Typescript docs. This is the setup I roll with for my component library:

```
{ "compilerOptions": { "outDir": "build/lib", "module": "commonjs", "target": "es5", "lib": ["es5", "es6",
"es7", "es2017", "dom"], "sourceMap": true, "allowJs": false, "jsx": "react", "moduleResolution": "node",
"rootDir": "src", "baseUrl": "src", "forceConsistentCasingInFileNames": true, "noImplicitReturns": true,
"strict": true, "esModuleInterop": true, "suppressImplicitAnyIndexErrors": true, "noUnusedLocals": true,
"declaration": true, "allowSyntheticDefaultImports": true, "experimentalDecorators": true }, "include":
["src/**/*"], "exclude": ["node_modules", "build", "scripts"] }
```

Please open an issue and discuss if there are better recommended choices for React.

Selected flags and why we like them:

- `esModuleInterop`: disables namespace imports (`import * as foo from "foo"`) and enables CJS/AMD/UMD style imports (`import fs from "fs"`)
- `strict`: `strictPropertyInitialization` forces you to initialize class properties or explicitly declare that they can be undefined. You can opt out of this with a definite assignment assertion.

Troubleshooting Handbook: Bugs in official typings

If you run into bugs with your library's official typings, you can copy them locally and tell TypeScript to use your local version using the "paths" field. In your `tsconfig.json`:

```
{ "compilerOptions": { "paths": { "mobx-react": ["../typings/modules/mobx-react"] } } }
```

Thanks to @adamrackis for the tip.

If you just need to add an interface, or add missing members to an existing interface, you don't need to copy the whole typing package. Instead, you can use declaration merging:

```
// my-typings.ts declare module 'plotly.js' { interface PlotlyHTMLElement { removeAllListeners(): void; }
} // MyComponent.tsx import { PlotlyHTMLElement } from 'plotly.js'; import './my-typings'; const f = (e:
PlotlyHTMLElement) => { e.removeAllListeners(); };
```

Recommended React + TypeScript codebases to learn from

- https://github.com/jaredpalmer/formik
- https://github.com/jaredpalmer/react-fns
- https://github.com/palantir/blueprint
- https://github.com/Shopify/polaris
- https://github.com/NullVoxPopuli/react-vs-ember/tree/master/testing/react
- https://github.com/artsy/reaction
- https://github.com/benawad/codeponder (with coding livestream!)

- https://github.com/artsy/emission (React Native)

React Boilerplates:

- @jpavon/react-scripts-ts alternative react-scripts with all TypeScript features using ts-loader
- webpack config tool is a visual tool for creating webpack projects with React and TypeScript
- https://github.com/innFactory/create-react-app-material-typescript-redux ready to go template with Material-UI, routing and Redux

React Native Boilerplates: *contributed by @spoeck*

- https://github.com/GeekyAnts/react-native-seed
- https://github.com/lopezjurip/ReactNativeTS
- https://github.com/emin93/react-native-template-typescript
- https://github.com/Microsoft/TypeScript-React-Native-Starter

Editor Tooling and Integration

- VSCode
  - swyx's VSCode Extension: https://github.com/sw-yx/swyx-react-typescript-snippets
  - amVim: https://marketplace.visualstudio.com/items?itemName=auiworks.amvim
- VIM
  - https://github.com/Quramy/tsuquyomi
  - nvim-typescript?
  - https://github.com/leafgarland/typescript-vim
  - peitalin/vim-jsx-typescript
  - NeoVim: https://github.com/neoclide/coc.nvim
  - other discussion: https://mobile.twitter.com/ryanflorence/status/1085715595994095620

Other React + TypeScript resources

- me! https://twitter.com/swyx
- https://github.com/piotrwitek/react-redux-typescript-guide - HIGHLY HIGHLY RECOMMENDED, i wrote this repo before knowing about this one, this has a lot of stuff I don't cover, including REDUX and JEST.
- Ultimate React Component Patterns with TypeScript 2.8
- Basarat's TypeScript gitbook has a React section with an Egghead.io course as well.
- Palmer Group's Typescript + React Guidelines as well as Jared's other work like disco.chat
- TypeScript React Starter Template by Microsoft A starter template for TypeScript and React with a detailed README describing how to use the two together. Note: this doesnt seem to be frequently updated anymore.
- Brian Holt's Intermediate React course on Frontend Masters (paid) - Converting App To Typescript Section
- Typescript conversion:
  - Lyft's React-To-Typescript conversion CLI
  - Gustav Wengel's blogpost - converting a React codebase to Typescript
  - Microsoft React Typescript conversion guide
- You?.

Recommended React + TypeScript talks

- Please help contribute this new section!

Time to Really Learn TypeScript

Believe it or not, we have only barely introduced TypeScript here in this cheatsheet. There is a whole world of generic type logic that you will eventually get into, however it becomes far less dealing with React than just getting good at TypeScript so it is out

of scope here. But at least you can get productive in React now :)

It is worth mentioning some resources to help you get started:

- Anders Hejlsberg's overview of TS: https://www.youtube.com/watch?v=ET4kT88JRXs
- Marius Schultz: https://blog.mariusschulz.com/series/typescript-evolution with an Egghead.io course
- Basarat's Deep Dive: https://basarat.gitbooks.io/typescript/
- Rares Matei: Egghead.io course's advanced Typescript course on Egghead.io is great for newer typescript features and practical type logic applications (e.g. recursively making all properties of a type `readonly`)

My question isn't answered here!

- Check out the Advanced Cheatsheet
- File an issue.