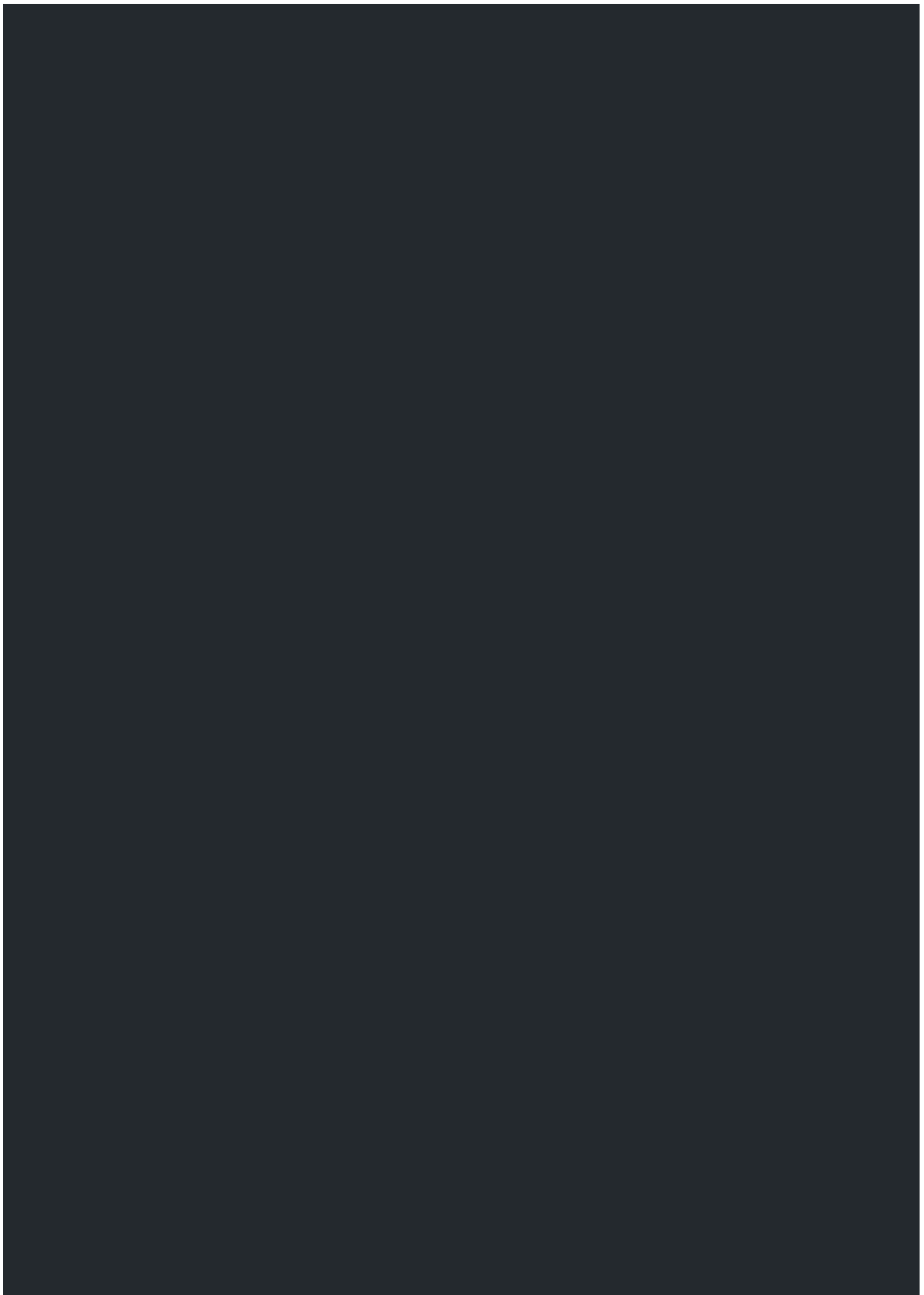
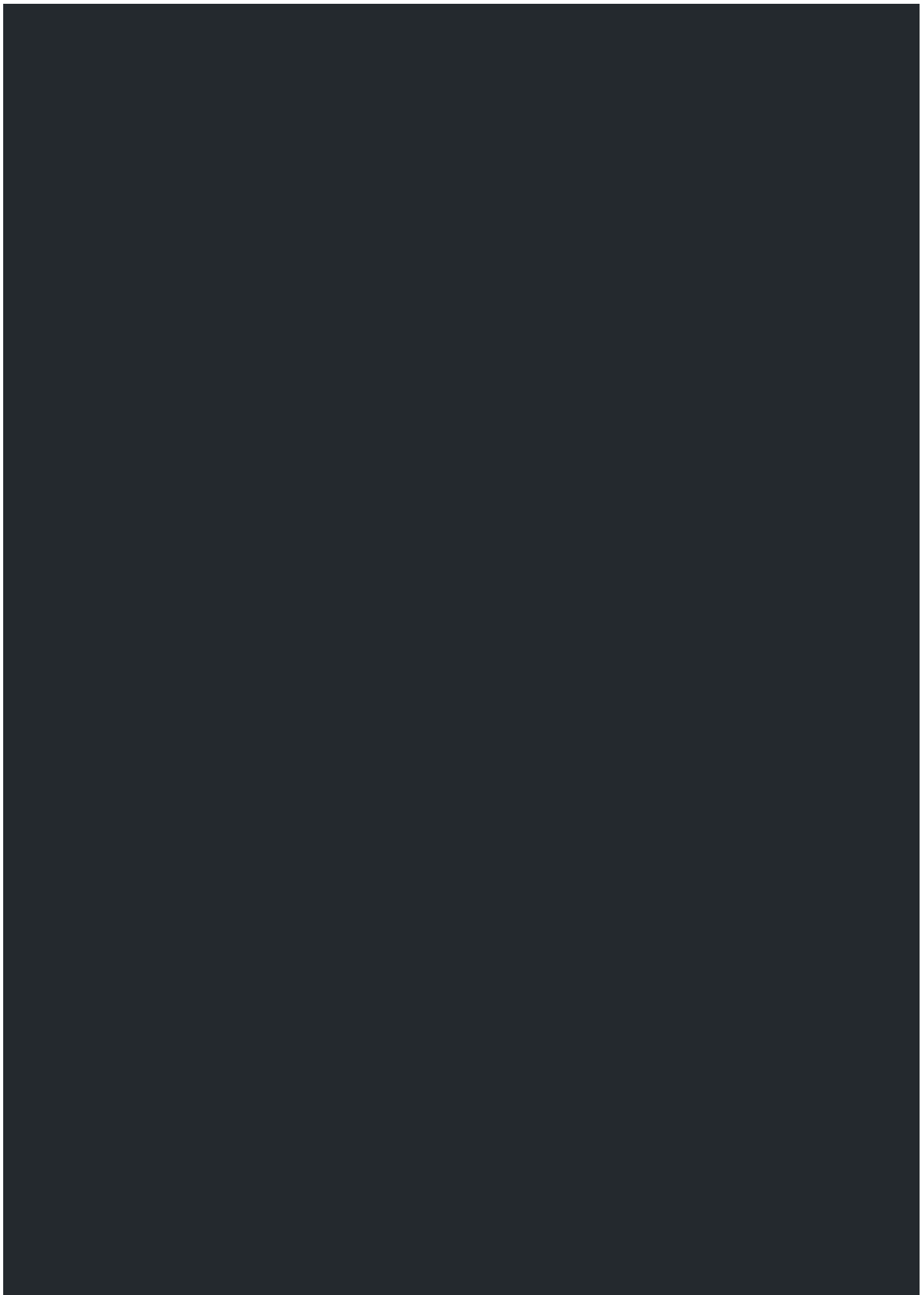
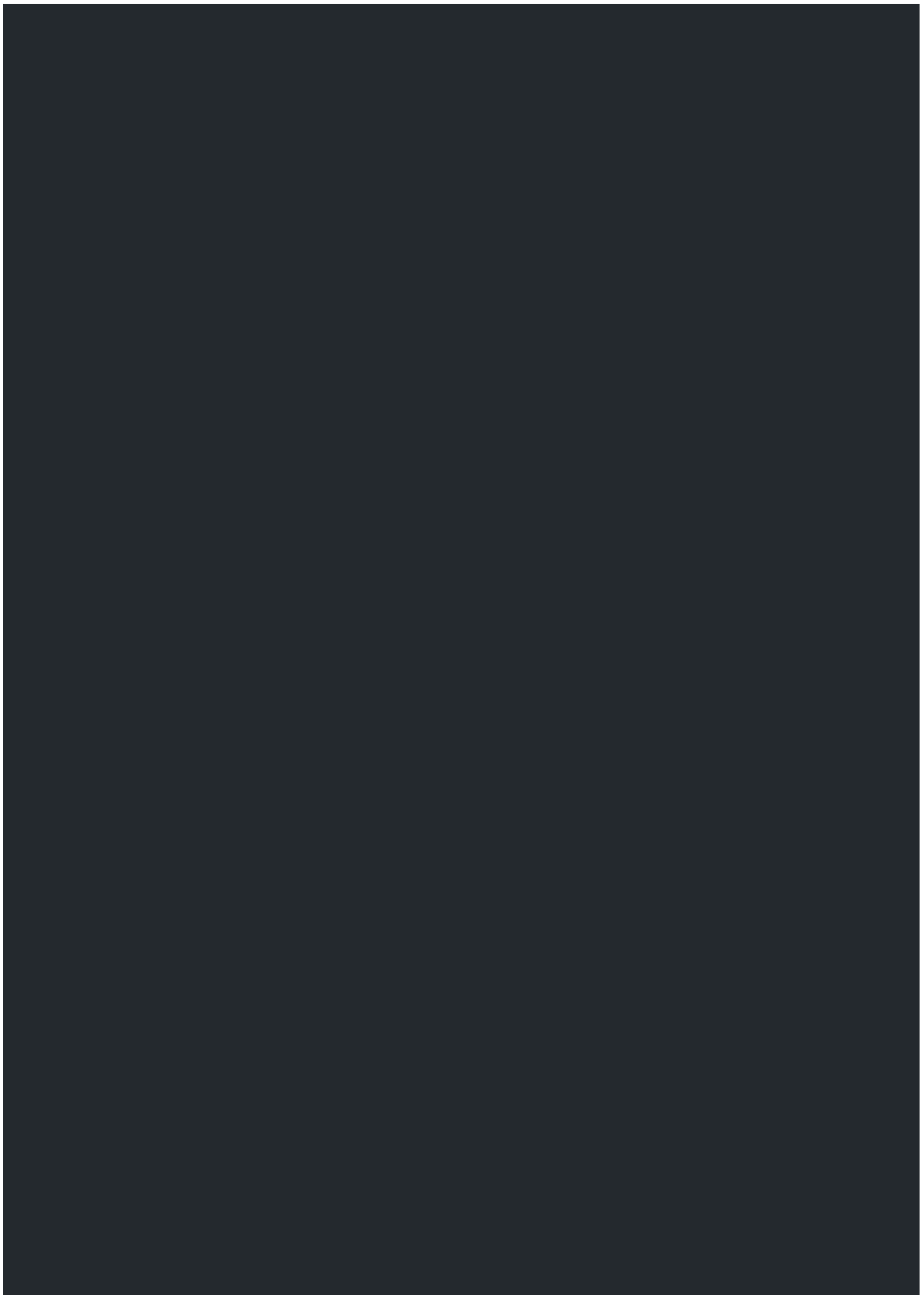


Enterprise

Marketplace







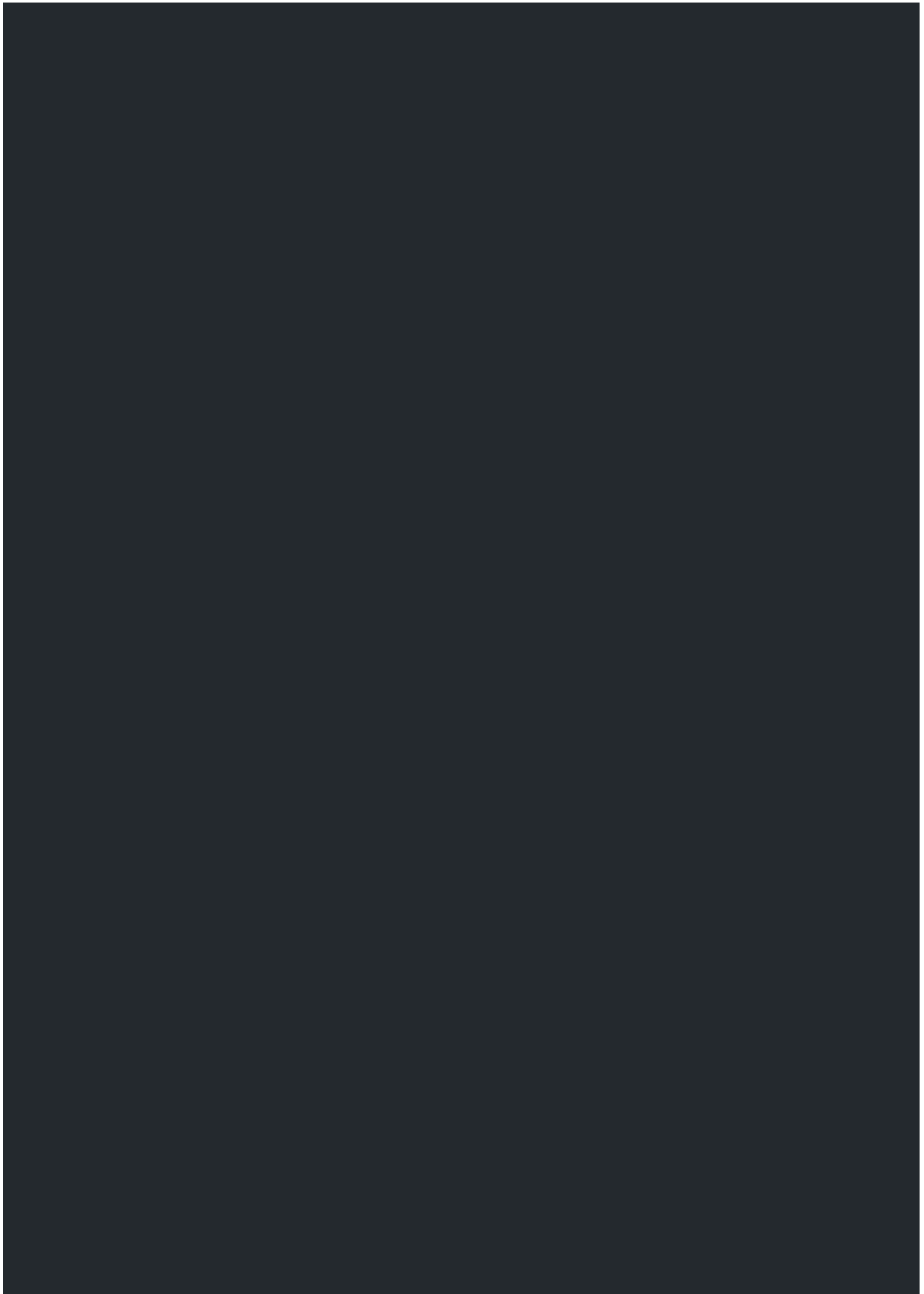


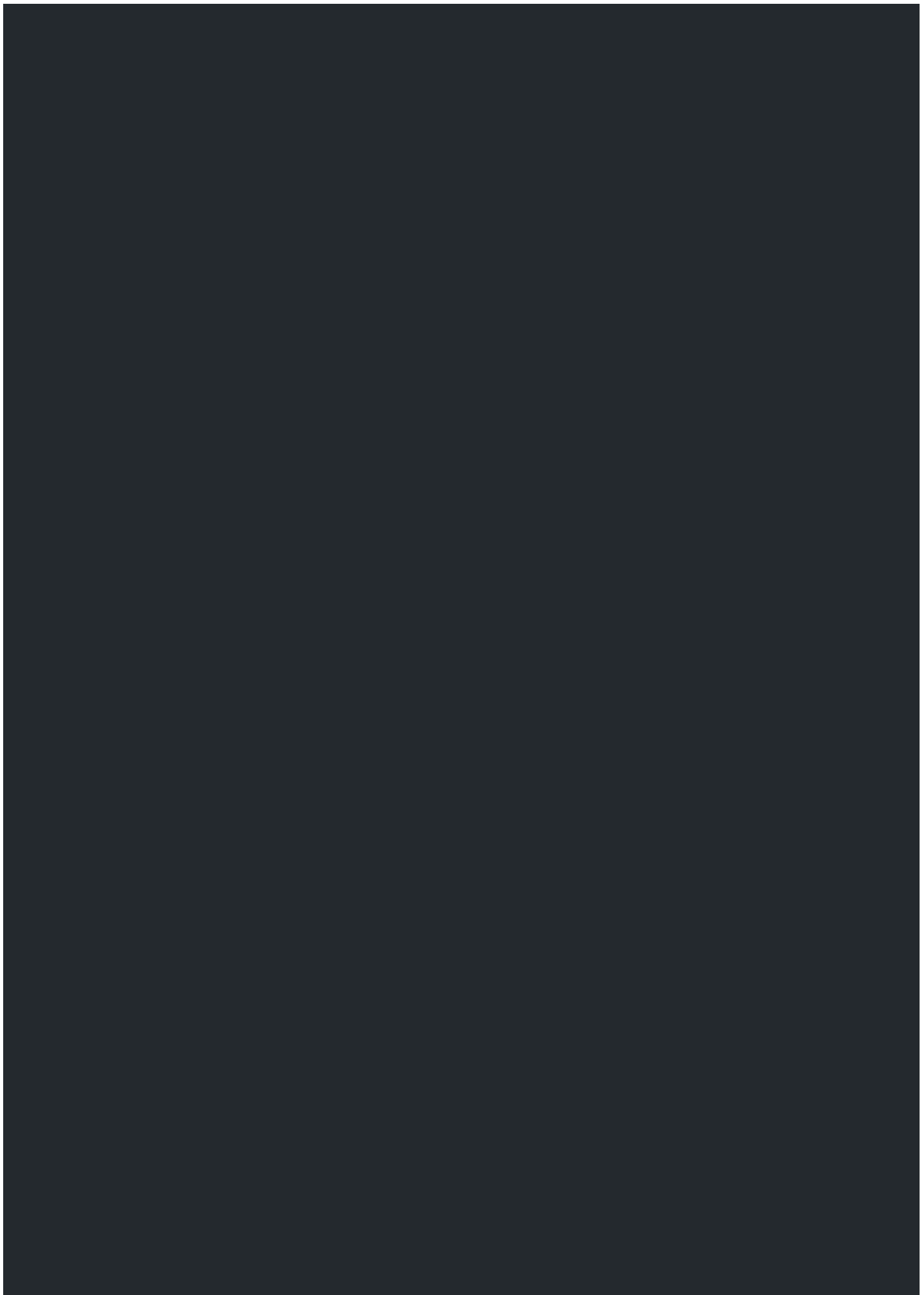
Search

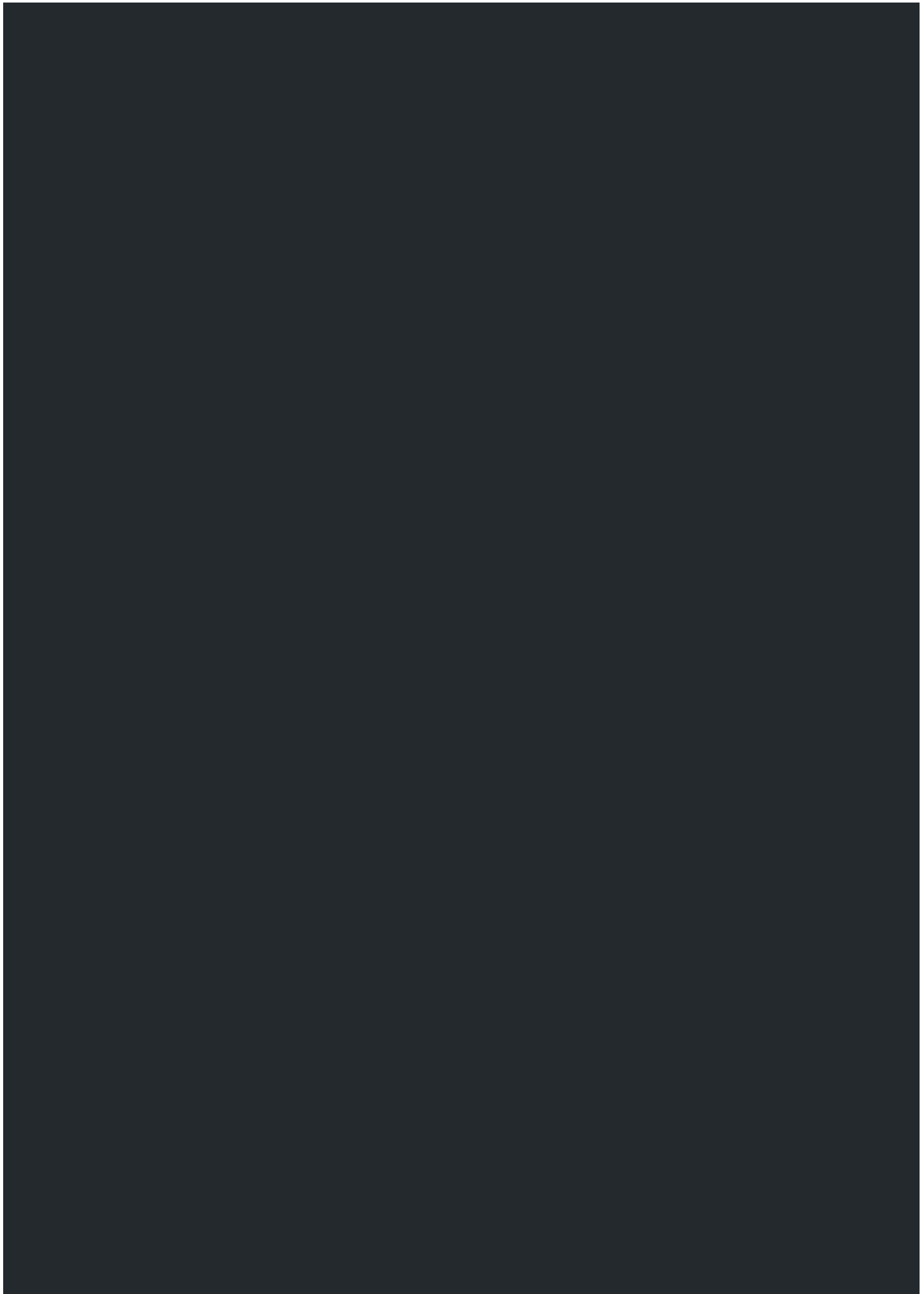


Sign in

Sign up











## Join GitHub today

GitHub is home to over 31 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)

Branch: master react-typescript-cheatsheet / ADVANCED.md

[Find file](#) [Copy path](#)

sw-yx add warning notice for tslint

18009b2 2 days ago

4 contributors

971 lines (731 sloc) 32.1 KB

Raw

Blame

History



Cheatsheets for experienced React developers getting started with TypeScript

[Basic](#) | [Advanced](#) | [Migrating](#) | [HOC](#) | [中文翻译](#) | [Contribute!](#) | [Ask!](#)

## Advanced Cheatsheet

This Advanced Cheatsheet 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.

### Advanced Cheatsheet Table of Contents

- ▶ [Expand Table of Contents](#)

## Section 0: Utility Types

Handy Utility Types used in the rest of this cheatsheet, or commonly used with React+TS apps, with explanation on what they do and how they can help. We will assume knowledge of [mapped types and conditional types](#) like `Exclude<T, U>` and `ReturnType<T>` but try to build progressively upon them.

- ▶ `Omit<T, K extends keyof T>`: Subtract keys from one interface from the other.
- ▶ `Optionalize<T extends K, K>`: Remove from T the keys that are in common with K
- ▶ `Nullable<T>` or `Maybe<T>`: Make a Type into a Maybe Type
- ▶ `Dictionary<T>`: Dictionary of string, value pairs

There also exist helper type libraries:

- [utility-types](#)

- [type-zoo](#)
- [typesafe-actions](#)

Something to add? [File an issue](#). We respect the fact that naming and selection of examples here is arbitrary as the possible space is infinite.

## Section 1: Advanced Guides

### Higher Order Components (HoCs)

Sometimes you want a simple way to inject props from somewhere else (either a global store or a provider) and don't want to continually pass down the props for it. Context is great for it, but then the values from the context can only be used in your `render` function. A HoC will provide these values as props.

The injected props

```
interface WithThemeProps { primaryColor: string; }
```

Usage in the component

The goal is to have the props available on the interface for the component, but subtracted out for the consumers of the component when wrapped in the HoC.

```
interface Props extends WithThemeProps { children: ReactNode; } class MyButton extends Component<Props> {
  public render() { // Render an the element using the theme and other props. } private someInternalMethod()
  { // The theme values are also available as props here. } } export default withTheme(MyButton);
```

Consuming the Component

Now when consuming the component you can omit the `primaryColor` prop or override the one provided through context.

```
<MyButton>Hello button</MyButton> // Valid <MyButton primaryColor="#333">Hello Button</MyButton> // Also
valid
```

Declaring the HoC

The actual HoC.

```
export function withTheme<T extends WithThemeProps> = WithThemeProps>( WrappedComponent:
React.ComponentType<T> ) { // Try to create a nice displayName for React Dev Tools. const displayName =
WrappedComponent.displayName || WrappedComponent.name || 'Component'; // Creating the inner component. The
calculated Props type here is the where the magic happens. return class ComponentWithTheme extends
React.Component< Optionalize<T, WithThemeProps> > { public static displayName =
`withPages(${displayName})`; public render() { // Fetch the props you want inject. This could be done with
context instead. const themeProps = getThemePropsFromSomewhere(); // this.props comes afterwards so the
can override the default ones. return <WrappedComponent {...themeProps} {...this.props as T} />; } } }
```

Note that the `{...this.props as T}` assertion is needed because of a current bug in TS 3.2

<https://github.com/Microsoft/TypeScript/issues/28938#issuecomment-450636046>

Here is a more advanced example of a dynamic higher order component that bases some of its parameters on the props of the component being passed in:

```
// inject static values to a component so that they're always provided export function inject<TProps,
TInjectedKeys extends keyof TProps>( Component: React.JSXElementConstructor<TProps>, injector:
Pick<TProps, TInjectedKeys> ) { return function Injected(props: Omit<TProps, TInjectedKeys>) { return
<Component {...props as TProps} {...injector} />; } }
```

Using `forwardRef`

For "true" reusability you should also consider exposing a ref for your HOC. You can use `React.forwardRef<Ref, Props>` as documented in [the basic cheatsheet](#), but we are interested in more real world examples. [Here is a nice example in practice](#) from @OliverJASH.

## Render Props

Sometimes you will want to write a function that can take a React element or a string or something else as a prop. The best Type to use for such a situation is `React.ReactNode` which fits anywhere a normal, well, React Node would fit:

```
export interface Props { label?: React.ReactNode; children: React.ReactNode; } export const Card = (props: Props) => { return ( <div> {props.label} && <div>{props.label}</div> } {props.children} </div> ); };
```

If you are using a function-as-a-child render prop:

```
export interface Props { children: (foo: string) => React.ReactNode; }
```

Something to add? [File an issue](#).

## as props (passing a component to be rendered)

`ElementType` is pretty useful to cover most types that can be passed to `createElement` e.g.

```
function PassThrough(props: { as: ElementType<any> }) { const { as: Component } = props; return <Component />; }
```

Thanks @eps1lon for this

## Typing a Component that Accepts Different Props

Components, and JSX in general, are analogous to functions. When a component can render differently based on their props, it's similar to how a function can be overloaded to have multiple call signatures. In the same way, you can overload a function component's call signature to list all of its different "versions".

A very common use case for this is to render something as either a button or an anchor, based on if it receives a `href` attribute.

```
type ButtonProps = JSX.IntrinsicElements['button']; type AnchorProps = JSX.IntrinsicElements['a']; // optionally use a custom type guard function isPropsForAnchorElement( props: ButtonProps | AnchorProps ): props is AnchorProps { return 'href' in props; } function Clickable(props: ButtonProps): JSX.Element; function Clickable(props: AnchorProps): JSX.Element; function Clickable(props: ButtonProps | AnchorProps) { if (isPropsForAnchorElement(props)) { return <a {...props} />; } else { return <button {...props} />; } }
```

They don't even need to be completely different props, as long as they have at least one difference in properties:

```
type LinkProps = Omit<JSX.IntrinsicElements['a'], 'href'> & { to?: string }; function RouterLink(props: LinkProps): JSX.Element; function RouterLink(props: AnchorProps): JSX.Element; function RouterLink(props: LinkProps | AnchorProps) { if ('to' in props) { return <a {...props} />; } else { return <Link {...props} />; } }
```

► Approach: Generic Components

► Approach: Composition

## Props: One or the Other but not Both

Use the `in` keyword, function overloading, and union types to make components that take either one or another sets of props,

but not both:

```
type Props1 = { foo: string }; type Props2 = { bar: string }; function MyComponent(props: Props1):
JSX.Element; function MyComponent(props: Props2): JSX.Element; function MyComponent(props: Props1 |
Props2) { if ('foo' in props) { // props.bar // error return <div>{props.foo}</div>; } else { // props.foo
// error return <div>{props.bar}</div>; } } const UsageComponent: React.FC = () => ( <div> <MyComponent
foo="foo" /> <MyComponent bar="bar" /> { /* <MyComponent foo="foo" bar="bar"/> // invalid */ } </div> );
```

## Props: Must Pass Both

```
type OneOrAnother<T1, T2> = | (T1 & { [K in keyof T2]?: undefined }) | (T2 & { [K in keyof T1]?: undefined
}); type Props = OneOrAnother<{ a: string; b: string }, {}>; const a: Props = { a: 'a' }; // error const
b: Props = { b: 'b' }; // error const ab: Props = { a: 'a', b: 'b' }; // ok
```

Thanks [diegohaz](#)

## Omit attribute from a type

Sometimes when intersecting types, we want to define our own version of an attribute. For example, I want my component to have a `label`, but the type I am intersecting with also has a `label` attribute. Here's how to extract that out:

```
export interface Props { label: React.ReactNode; // this will conflict with the InputElement's label }
type Omit<T, K extends keyof T> = Pick<T, Exclude<keyof T, K>>; // usage export const Checkbox = ( props:
Props & Omit<React.HTMLProps<HTMLInputElement>, 'label'> ) => { const { label } = props; return ( <div
className="Checkbox"> <label className="Checkbox-label"> <input type="checkbox" {...props} /> </label>
<span>{label}</span> </div> ); };
```

## Type Zoo

As you can see from the Omit example above, you can write significant logic in your types as well. [type-zoo](#) is a nice toolkit of operators you may wish to check out (includes Omit), as well as [utility-types](#) (especially for those migrating from Flow).

## Extracting Prop Types of a Component

(Contributed by [@ferdaber](#))

There are a lot of places where you want to reuse some slices of props because of prop drilling, so you can either export the props type as part of the module or extract them (either way works).

The advantage of extracting the prop types is that you won't need to export everything, and a refactor of the source of truth component will propagate to all consuming components.

```
import { ComponentProps, JSXElementConstructor } from 'react'; // goes one step further and resolves with
propTypes and defaultProps properties type ApparentComponentProps<C> = C extends
JSXElementConstructor<infer P> ? JSX.LibraryManagedAttributes<C, P> : ComponentProps<C>;
```

You can also use them to strongly type custom event handlers if they're not written at the call sites themselves (i.e. inlined with the JSX attribute):

```
// my-inner-component.tsx export function MyInnerComponent(props: { onSomeEvent( event: ComplexEventObj,
moreArgs: ComplexArgs ): SomeWeirdReturnType; }) { /* ... */ } // my-consuming-component.tsx export
function MyConsumingComponent() { // event and moreArgs are contextually typed along with the return value
const theHandler: Props<typeof MyInnerComponent>['onSomeEvent'] = ( event, moreArgs ) => {}; return
<MyInnerComponent onSomeEvent={theHandler} />; }
```

## Third Party Libraries

Sometimes DefinitelyTyped can get it wrong, or isn't quite addressing your use case. You can declare your own file with the same interface name. Typescript will merge interfaces with the same name.

## Section 2: Useful Patterns by TypeScript Version

TypeScript Versions often introduce new ways to do things; this section helps current users of React + TypeScript upgrade TypeScript versions and explore patterns commonly used by TypeScript + React apps and libraries. This may have duplications with other sections; if you spot any discrepancies, [file an issue](#)!

TypeScript version guides before 2.9 are unwritten, please feel free to send a PR! Apart from official TS team communication we also recommend [Marius Schulz's blog for version notes](#).

### TypeScript 2.9

[\[Release Notes\]](#) | [\[Blog Post\]](#)

1. Type arguments for tagged template strings (e.g. `styled-components`):

```
export interface InputFormProps { foo: string; // this is understood inside the template string below }
export const InputForm = styledInput<InputFormProps>` color: ${({ themeName }) => (themeName === 'dark' ?
'black' : 'white')}; border-color: ${({ foo }) => (foo ? 'red' : 'black')}; `;
```

2. JSX Generics

<https://github.com/Microsoft/TypeScript/pull/22415>

Helps with typing/using generic components:

```
// instead of <Formik render={({props: FormikProps<Values>} => ...})/> // usage <Formik<Values> render=
{props => ...}/> <MyComponent<number> data={12} />
```

More info: <https://github.com/basarat/typescript-book/blob/master/docs/jsx/react.md#react-jsx-tip-generic-components>

### TypeScript 3.0

[\[Release Notes\]](#) | [\[Blog Post\]](#)

1. Typed rest parameters for writing arguments of variable length:

```
// `rest` accepts any number of strings - even none! function foo(...rest: string[]) { // ... }
foo('hello'); // works foo('hello', 'world'); // also works
```

2. Support for `propTypes` and `static defaultProps` in JSX using `LibraryManagedAttributes`:

```
export interface Props { name: string; } export class Greet extends React.Component<Props> { render() {
const { name } = this.props; return <div>Hello ${name.toUpperCase()}!</div>; } static defaultProps = {
name: 'world' }; } // Type-checks! No type assertions needed! let el = <Greet />;
```

3. new `Unknown` type

For typing API's to force type checks - not specifically React related, however very handy for dealing with API responses:

```
interface IComment { date: Date; message: string; } interface IDataService1 { getData(): any; } let
service1: IDataService1; const response = service1.getData(); response.a.b.c.d; // RUNTIME ERROR // -----
compare with ----- interface IDataService2 { getData(): unknown; // ooo } let service2: IDataService2;
const response2 = service2.getData(); // response2.a.b.c.d; // COMPILE TIME ERROR if you do this if
(typeof response === 'string') { console.log(response.toUpperCase()); // `response` now has type 'string'
}
```

You can also assert a type, or use a type guard against an `unknown` type. This is better than resorting to `any`.

## TypeScript 3.1

[\[Release Notes\]](#) | [\[Blog Post\]](#)

1. Properties declarations on functions

Attaching properties to functions like this "just works" now:

```
export const FooComponent => ({ name }) => ( <div>Hello! I am {name}</div> ); FooComponent.defaultProps = { name: "swyx", };
```

## TypeScript 3.2

[\[Release Notes\]](#) | [\[Blog Post\]](#)

nothing specifically React related.

## TypeScript 3.3

[\[Release Notes\]](#) | [\[Blog Post\]](#)

nothing specifically React related.

## TypeScript Roadmap

<https://github.com/Microsoft/TypeScript/wiki/Roadmap>

## Section 3: Misc. Concerns

Sometimes writing React isn't just about React. While we don't focus on other libraries like Redux (see below for more on that), here are some tips on other common concerns when making apps with React + TypeScript.

## Writing TypeScript Libraries instead of Apps

`propTypes` may seem unnecessary with TypeScript, especially when building React + TypeScript apps, but they are still relevant when writing libraries which may be used by developers working in Javascript.

```
interface IMyComponentProps { autoHeight: boolean; secondProp: number; } export class MyComponent extends React.Component<IMyComponentProps, {}> { static propTypes = { autoHeight: PropTypes.bool, secondProp: PropTypes.number.isRequired }; }
```

Something to add? [File an issue.](#)

## Commenting Components

Typescript uses [TSDoc](#), a variant of JSDoc for Typescript. This is very handy for writing component libraries and having useful descriptions pop up in autocomplete and other tooling (like the [Docz PropsTable](#)). The main thing to remember is to use `/** YOUR_COMMENT_HERE */` syntax in the line just above whatever you're annotating.

```
import React from 'react'; interface MyProps { /** Description of prop "label". * @default foobar */ label?: string; } /** * General component description in JSDoc format. Markdown is *supported*. */ export default function MyComponent({ label = 'foobar' }: MyProps) { return <div>Hello world {label}</div>; }
```

Something to add? [File an issue.](#)

# Design System Development

I do like [Docz](#) which takes basically [1 line of config](#) to accept Typescript. However it is newer and has a few more rough edges (many breaking changes since it is still < v1.0)

For developing with Storybook, read the docs I wrote over here: <https://storybook.js.org/configurations/typescript-config/>. This includes automatic proptype documentation generation, which is awesome :)

Something to add? [File an issue](#).

## Migrating From Flow

You should check out large projects that are migrating from flow to pick up concerns and tips:

- [Jest](#)
- [Expo](#)
- [React-beautiful-dnd](#)
- [Storybook](#)
- [VueJS](#)

Useful libraries:


- <https://github.com/bcherny/flow-to-typescript>
- <https://github.com/piotrwitek/utility-types>.

If you have specific advice in this area, please file a PR!

Something to add? [File an issue](#).

## Prettier + TSLint

Contributed by: [@azdanov](#)

 Note that [TSLint is now in maintenance and you should try to use ESLint instead](#). The rest of this section is potentially outdated.


To use prettier with TSLint you will need [tslint-config-prettier](#) which disables all the conflicting rules and optionally [tslint-plugin-prettier](#) which will highlight differences as TSLint issues.

Example configuration:

tslint.json	.prettierrc
<pre>{ "rulesDirectory": ["tslint-plugin-prettier"],   "extends": [ "tslint:recommended", "tslint-config-prettier" ],   "linterOptions": { "exclude": ["node_modules/**/*.ts"] },   "rules": { "prettier": true } }</pre>	<pre>{ "printWidth": 89, "tabWidth": 2,   "useTabs": false, "semi": true,   "singleQuote": true, "trailingComma":   "all", "bracketSpacing": true,   "jsxBracketSameLine": false }</pre>

An example github repository with a project showing how to integrate [prettier + tslint + create-react-app-ts](#).

## ESLint + TSLint

 This is an evolving topic. [typescript-eslint-parser](#) is no longer maintained and [work has recently begun on typescript-eslint in the ESLint community](#) to bring ESLint up to full parity and interop with TSLint. The rest of this section is potentially outdated.

Why use ESLint with/over TSLint? ESLint ecosystem is rich, with lots of different plugins and config files, whereas TSLint tend to lag behind in some areas.

To remedy this nuisance there is an [typescript-eslint-parser](#) which tries to bridge the differences between javascript and



typescript. It still has some rough corners, but can provide consistent assistance with certain plugins.

Usage	.eslintrc
<pre>// Install: npm i -D typescript- eslint-parser  // And in your ESLint configuration file:  "parser": "typescript- eslint- parser"</pre>	<pre>{ "extends": [ "airbnb", "prettier", "prettier/react", "plugin:prettier/recommended", "plugin:jest/recommended", "plugin:unicorn/recommended" ], "plugins": ["prettier", "jest", "unicorn"], "parserOptions": { "sourceType": "module", "ecmaFeatures": { "jsx": true } }, "env": { "es6": true, "browser": true, "jest": true }, "settings": { "import/resolver": { "node": { "extensions": [".js", ".jsx", ".ts", ".tsx"] } } }, "overrides": [ { "files": ["**/*.ts", "**/*.tsx"], "parser": "typescript-eslint- parser", "rules": { "no-undef": "off" } } ] }</pre>

An example github repository with a project showing how to integrate [eslint + tslint + create-react-app-ts](#).

## Working with Non-TypeScript Libraries (writing your own index.d.ts)

Lets say you want to use `de-indent`, but it isn't typed or on DefinitelyTyped. You get an error like this:

```
[ts]
Could not find a declaration file for module 'de-indent'. '/Users/swyx/Work/react-sfc-loader/node_modules/de-indent'
Try `npm install @types/de-indent` if it exists or add a new declaration (.d.ts) file containing `declare module 'de-indent'`
```

So create a `.d.ts` file anywhere in your project with the module definition:

```
// de-indent.d.ts declare module 'de-indent' { function deindent(): void export = deindent // default
export }
```

► Further Discussion

## Section 4: @types/react and @types/react-dom APIs

The `@types` typings export both "public" types meant for your use as well as "private" types that are for internal use.

### @types/react

[Link to .d.ts](#)

Namespace: React

Most Commonly Used Interfaces and Types

- `ReactNode` - anything that is renderable *inside* of JSX, this is NOT the same as what can be rendered by a component!
- `Component` - base class of all class-based components
- `PureComponent` - base class for all class-based optimized components
- `FC`, `FunctionComponent` - a complete interface for function components, often used to type external components instead of typing your own
- `CSSProperties` - used to type style objects
- all events: used to type event handlers
- all event handlers: used to type event handlers
- all consts: `Children`, `Fragment`, ... are all public and reflect the React runtime namespace

## Not Commonly Used but Good to know

- `Ref` - used to type `innerRef`
- `ElementType` - used for higher order components or operations on components
- `ComponentType` - used for higher order components where you don't specifically deal with the intrinsic components
- `ReactPortal` - used if you specifically need to type a prop as a portal, otherwise it is part of `ReactNode`
- `ComponentClass` - a complete interface for the produced constructor function of a class declaration that extends `Component`, often used to type external components instead of typing your own
- `JSXElementConstructor` - anything that TypeScript considers to be a valid thing that can go into the opening tag of a JSX expression
- `ComponentProps` - props of a component
- `ComponentPropsWithRef` - props of a component where if it is a class-based component it will replace the `ref` prop with its own instance type
- `ComponentPropsWithoutRef` - props of a component without its `ref` prop
- all methods: `createElement`, `cloneElement`, ... are all public and reflect the React runtime API

[@Ferdaber's note](#): I discourage the use of most `...Element` types because of how black-boxy `JSX.Element` is. You should almost always assume that anything produced by `React.createElement` is the base type `React.ReactElement`.

## Namespace: JSX

- `Element` - the type of any JSX expression
- `LibraryManagedAttributes` - used to resolve static `defaultProps` and `propTypes` with the internal props type of a component
- `IntrinsicElements` - every possible built-in component that can be typed in as a lowercase tag name in JSX

## Don't use/Internal/Deprecated

Anything not listed above is considered an internal type and not public. If you're not sure you can check out the source of `@types/react`. The types are annotated accordingly.

- `SFCElement`
- `SFC`
- `ComponentState`
- `LegacyRef`
- `StatelessComponent`
- `ReactType`

## @types/react-dom

To be written

## My question isn't answered here!

- [File an issue.](#)

