StoryBook

1. Introduction

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* StoryBook is required only during development
* StoryBook has fulfilled its job of being a development environment and playground for UI components

1. Installation
2. Installation

* **npx sb init** 🡺 **Help to initialize StoryBook in my project**
* **npx run storybook** 🡺 **Run storybook.**

When we install storybook storybook it will create a **.storybook** folder in our project in this folder we have **main.js** 🡪It is the configuration file for storybook itself. and also, we have **preview.js** 🡪It is the configuration file for the stories that I write.

.src/stories ???

1. Difference between preview.ts and main.ts

In Storybook, **main.ts** and **preview.js** serve different purposes:

1. **main.ts**: This file is typically used for configuring Storybook. It's the entry point for configuring Storybook's environment, where you can define settings like addons, decorators, and parameters for your Storybook instance. **main.ts** allows you to customize the behavior of Storybook itself.
2. **preview.js**: On the other hand, **preview.js** is more focused on configuring the preview of your components within Storybook. It allows you to add global decorators, set up global parameters, import global styles, or perform any other necessary setup specifically related to the rendering and display of your components in Storybook's UI.

In summary, **main.ts** is for configuring Storybook itself, while **preview.js** is for configuring how your components are displayed and interacted with within Storybook. They serve different purposes but work together to provide a customized and efficient development environment for your components.

1. StoryBook
2. Writing stories.

Writing your stories in [TypeScript](https://www.typescriptlang.org/) makes you more productive. You don't have to jump between files to look up component props. Your code editor will alert you about missing required props and even autocomplete prop values, just like when using your components within your app. Plus, Storybook infers those component types to auto-generate the [Controls](https://storybook.js.org/docs/api/doc-block-controls) table.

Storybook has built-in TypeScript support, so you can get started with zero configuration required.

## Typing stories with Meta and StoryObj

When writing stories, there are two aspects that are helpful to type. The first is the [component meta](https://storybook.js.org/docs/writing-stories/#default-export), which describes and configures the component and its stories. In a [CSF file](https://storybook.js.org/docs/api/csf), this is the default export. The second is the [stories themselves](https://storybook.js.org/docs/writing-stories/#defining-stories).

Storybook provides utility types for each of these, named Meta and StoryObj. Here's an example CSF file using those types:

import type { Meta, StoryObj } from '@storybook/react';  
import {*Button*} from "@mui/material";  
  
  
const *meta* = {  
 title: 'Button',  
 component: *Button*,  
} satisfies Meta<typeof *Button*>;  
  
export default *meta*;  
type Story = StoryObj<typeof *meta*>;  
  
export const *Text*:Story={render:()=> <Button variant="text">Text</Button>}  
  
export const *Contained*:Story={render:()=> <Button variant="contained">Contained</Button>}  
  
export const *Outlined*:Story={render:()=> <Button variant="outlined">Outlined</Button>}

### Props type parameter

Meta and StoryObj types are both [generics](https://www.typescriptlang.org/docs/handbook/2/generics.html#working-with-generic-type-variables), so you can provide them with an optional prop type parameter for the component type or the component's props type (e.g., the typeof Button portion of Meta<typeof Button>). By doing so, TypeScript will prevent you from defining an invalid arg, and all [decorators](https://storybook.js.org/docs/writing-stories/decorators), [play functions](https://storybook.js.org/docs/writing-stories/play-function), or [loaders](https://storybook.js.org/docs/writing-stories/loaders) will type their function arguments.

The example above passes a component type. See [**Typing custom args**](https://storybook.js.org/docs/writing-stories/typescript#typing-custom-args) for an example of passing a props type.

## Using satisfies for better type safety

If you are using TypeScript 4.9+, you can take advantage of the new [satisfies](https://www.typescriptlang.org/docs/handbook/release-notes/typescript-4-9.html) operator to get stricter type checking. Now you will receive type errors for missing required args, not just invalid ones.

Using satisfies to apply a story's type helps maintain type safety when sharing a [play function](https://storybook.js.org/docs/writing-stories/play-function) across stories. Without it, TypeScript will throw an error that the play function may be undefined. The satisfies operator enables TypeScript to infer whether the play function is defined or not.

Finally, use of satisfies allows you to pass typeof meta to the StoryObj generic. This informs TypeScript of the connection between the meta and StoryObj types, which allows it to infer the args type from the meta type. In other words, TypeScript will understand that args can be defined both at the story and meta level and won't throw an error when a required arg is defined at the meta level, but not at the story level.

## Typing custom args

Sometimes stories need to define args that aren’t included in the component's props. For this case, you can use an [intersection type](https://www.typescriptlang.org/docs/handbook/2/objects.html#intersection-types) to combine a component's props type and your custom args' type. For example, here's how you could use a footer arg to populate a child component:

import type { Meta, StoryObj } from '@storybook/react';  
  
import { Page } from './Page';  
  
type PagePropsAndCustomArgs = *React*.ComponentProps<typeof Page> & { footer?: string };  
  
const *meta*: Meta<PagePropsAndCustomArgs> = {  
 component: Page,  
 render: ({ footer, ...args }) => (  
 <Page {...args}>  
 <footer>{footer}</footer>  
 </Page>  
 ),  
};  
export default *meta*;  
  
type Story = StoryObj<PagePropsAndCustomArgs>;  
  
export const *CustomFooter*: Story = {  
 args: {  
 footer: 'Built with Storybook',  
 },  
};

1. Grouping stories.
2. import type { Meta, StoryObj } from '@storybook/react';  
   import {*Button*} from "@mui/material";  
     
     
   const *meta* = {  
    title: 'Form/Control/Button',  
    component: *Button*,  
   } satisfies Meta<typeof *Button*>;  
     
   export default *meta*;  
   type Story = StoryObj<typeof *meta*>;  
     
   export const *Text*:Story={render:()=> <Button variant="text">Text</Button>}  
     
   export const *Contained*:Story={render:()=> <Button variant="contained">Contained</Button>}  
     
   export const *Outlined*:Story={render:()=> <Button variant="outlined">Outlined</Button>}

import type { Meta, StoryObj } from '@storybook/react';  
import {TextField} from "@mui/material";  
  
  
const *meta* = {  
 title: 'Form/TextField',  
 component: TextField,  
} satisfies Meta<typeof TextField>;  
  
export default *meta*;  
type Story = StoryObj<typeof *meta*>;  
  
export const *Outlined*:Story={render:()=> <TextField id="outlined-basic" label="Outlined" variant="outlined" />}  
  
export const *Filled*:Story={render:()=><TextField id="filled-basic" label="Filled" variant="filled" />}  
  
export const *Standard*:Story={render:()=> <TextField id="standard-basic" label="Standard" variant="standard" />}

Une image contenant capture d’écran, texte

Description générée automatiquement

1. Rename the story.

export const *Outlined*:Story={ name:"OutlinedTextField", render:()=> <TextField id="outlined-basic" label="Outlined" variant="outlined" />}

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

1. Sorting all stories.

Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, document

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

1. Using Args

A story is a component with a set of arguments that define how the component should render. “Args” are Storybook’s mechanism for defining those arguments in a single JavaScript object. Args can be used to dynamically change props, slots, styles, inputs, etc. It allows Storybook and its addons to live edit components. You **do not** need to modify your underlying component code to use args.

When an arg’s value changes, the component re-renders, allowing you to interact with components in Storybook’s UI via addons that affect args.

Learn how and why to write stories in [the introduction](https://storybook.js.org/docs/writing-stories/#using-args). For details on how args work, read on.

## Args object

The args object can be defined at the [story](https://storybook.js.org/docs/writing-stories/args#story-args), [component](https://storybook.js.org/docs/writing-stories/args#component-args) and [global level](https://storybook.js.org/docs/writing-stories/args#global-args). It is a JSON serializable object composed of string keys with matching valid value types that can be passed into a component for your framework.

## Story args and Component args and Args composition

import type { Meta, StoryObj } from '@storybook/react';  
import {*Button*} from "@mui/material";  
  
  
const *meta* = {  
 title: 'Form/Control/Button',  
 component: *Button*,  
 args:{  
 variant:"contained",  
 children:"Button Children Global Args"  
 }  
} satisfies Meta<typeof *Button*>;  
  
export default *meta*;  
type Story = StoryObj<typeof *meta*>;  
  
export const *Text*:Story={render:()=> <Button variant="text">Text</Button>}  
  
export const *Contained*:Story={render:()=> <Button variant="contained">Contained</Button>}  
  
export const *Outlined*:Story={render:()=> <Button variant="outlined">Outlined</Button>}  
  
export const *TextArg*:Story={  
 args:{  
 variant:"text",  
 children:"Text Args"  
 }  
}  
export const *LongTextArg*:Story={  
 args:{  
 ...*TextArg*.*args*,  
 children:"Long Text Args"  
 }  
}  
  
export const *GlobalTextArg*:Story={  
 args:{  
 color:"secondary"  
 }  
}

Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Description générée automatiquement

Une image contenant texte, logiciel, Icône d’ordinateur, Page web

Description générée automatiquement

Une image contenant texte, logiciel, Icône d’ordinateur, Logiciel multimédia

Description générée automatiquement

## Global args

Une image contenant texte, capture d’écran, Police, Page web

Description générée automatiquement

## Setting args from within a story

Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement

## Mapping to complex arg values

Une image contenant texte, capture d’écran, Police, document

Description générée automatiquement

1. Decorators

A decorator is a way to wrap a story in extra “rendering” functionality. Many addons define decorators to augment your stories with extra rendering or gather details about how your story renders.

* 1. Add decorators just for one story file.

import React from 'react'

import Button from './Button'

export default {

title:'Form/Button',

component:Button,

args:{

children:'ButtonArgs'

},

decorators:[story=><div style={{display:"flex",justifyContent:"center",height:"100vh",alignItems:"center"}}>{story()}</div>]

}

export const Primary=()=> <Button variant='primary'>Primary</Button>

export const Secondary=()=> <Button variant='secondary'>Secondary</Button

Une image contenant capture d’écran, texte, logiciel

Description générée automatiquement

* 1. Add decorators for all stories files.

If we want to add decorators for all our stories file we can just add some code to my **.storybook/preview.js**

export const decorators=[story=><div style={{display:"flex",justifyContent:"center",height:"100vh",alignItems:"center"}}>{story()}</div>]

1. Addons

In Storybook, addons are additional functionalities or tools that extend the capabilities of the development environment. They enhance the workflow by providing features such as testing utilities, design documentation, or UI customization options. Addons help developers streamline the development process and improve collaboration by offering a wide range of features tailored to specific needs.

A major strength of Storybook are [addons](https://storybook.js.org/addons) that extend Storybook’s UI and behavior. Storybook ships by default with a set of “essential” addons that add to the initial user experience. There are many third-party addons as well as “official” addons developed by the Storybook core team.

* [Actions](https://storybook.js.org/docs/essentials/actions)
* [Backgrounds](https://storybook.js.org/docs/essentials/backgrounds)
* [Controls](https://storybook.js.org/docs/essentials/controls)
* [Docs](https://storybook.js.org/docs/writing-docs)
* [Highlight](https://storybook.js.org/docs/essentials/highlight)
* [Measure & outline](https://storybook.js.org/docs/essentials/measure-and-outline)
* [Toolbars & globals](https://storybook.js.org/docs/essentials/toolbars-and-globals)
* [Viewport](https://storybook.js.org/docs/essentials/viewport)
  1. **Actions**

The actions addon is used to display data received by event handler (callback) arguments in your stories.

**Une image contenant texte, logiciel, Icône d’ordinateur, Page web

Description générée automatiquement**

* 1. **Docs addon**

When you write component stories during development, you also create basic documentation to revisit later.

Storybook gives you tools to expand this essential documentation with prose and layout that feature your components and stories prominently. That allows you to create UI library usage guidelines, design system sites, and more.

**Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Description générée automatiquement**

* 1. **Viewport addon**

The Viewport toolbar item allows you to adjust the dimensions of the iframe your story is rendered in. It makes it easy to develop responsive UIs.

**Une image contenant texte, logiciel, Icône d’ordinateur, Logiciel multimédia

Description générée automatiquement**

* 1. **a11y Addon**

<https://www.youtube.com/watch?v=rNLL0SICr9w&t=3s&ab_channel=Chromatic>

Accessibility is the practice of making websites inclusive to all. That means supporting requirements such as: keyboard navigation, screen reader support, touch-friendly, usable color contrast, reduced motion, and zoom support.

Accessibility tests audit the rendered DOM against a set of heuristics based on [WCAG](https://www.w3.org/WAI/standards-guidelines/wcag/) rules and other industry-accepted best practices. They act as the first line of QA to catch blatant accessibility violations.

Une image contenant texte, capture d’écran, diagramme, Police

Description générée automatiquement

## Accessibility checks with a11y addon

Storybook provides an official [a11y addon](https://storybook.js.org/addons/@storybook/addon-a11y). Powered by Deque's [axe-core](https://github.com/dequelabs/axe-core), which automatically catches up to [57% of WCAG issues](https://www.deque.com/blog/automated-testing-study-identifies-57-percent-of-digital-accessibility-issues/).

### Set up the a11y addon

If you want to check accessibility for your stories using the [addon](https://storybook.js.org/addons/@storybook/addon-a11y/), you'll need to install it and add it to your Storybook.

Run the following command to install the addon.

1. Environment variables

You can use environment variables in Storybook to change its behavior in different “modes”. If you supply an environment variable prefixed with STORYBOOK\_, it will be available in process.env when using Webpack, or import.meta.env when using the Vite builder:

Une image contenant texte, Police, ligne, capture d’écran

Description générée automatiquement

Une image contenant texte, Police, ligne, capture d’écran

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, algèbre

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

1. Building storybook

Teams publish Storybook online to review and collaborate on works in progress. That allows developers, designers, PMs, and other stakeholders to check if the UI looks right without touching code or requiring a local dev environment.

## Build Storybook as a static web application

First, we'll need to build Storybook as a static web application. The functionality is already built-in and pre-configured for most supported frameworks. Others require a bit of customization (e.g., Angular). Run the following command inside your project's root directory:

Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement