

www.agilityio.com

Storybook Notes

[**Docs: Storybook**](https://storybook.js.org/docs/react/get-started/introduction)

**Chapter 1: Introduction to Storybook for React**

**Overview**

* The Storybook is a tool for UI development. It makes development faster and easier by isolating components. This allows you to work on one component at a time. You can develop entire UIs without needing to start up a complex dev stack, force certain data into your database, or navigate around your application
* Storybook helps you document components for reuse and automatically visually test your components to prevent bugs. Extend Storybook with an ecosystem of addons that help you do things like fine-tune responsive layouts or verify accessibility.

# Chapter 2: An Introduction to How to Write Stories

## Where to put stories

* A story captures the rendered state of a UI component. It’s a function that returns a component’s state given a set of arguments.
* A component’s stories are defined in a story file that lives alongside the component file. The story file is for development only, and it won't be included in your production bundle.
* Example:
* Button.js | ts | jsx | tsx | vue | svelte
* Button.stories.js | ts | jsx | tsx | mdx

## Component Story Format

* We define stories according to the Component Story Format (CSF), an ES6 module-based standard that is easy to write and portable between tools.
* The key ingredients are the default export that describes the component and named exports that describe the stories.

Default export

* The default export metadata controls how Storybook lists your stories and provides information used by addons

Defining stories

* Use the named exports of a CSF file to define your component’s stories. We recommend you use UpperCamelCase for your story exports

Rename stories

* You can rename any particular story you need.

How to write stories

* A story is a function that describes how to render a component. You can have multiple stories per component, and the simplest way to create stories is to render a component with different arguments multiple times.

Using args

* Refine this pattern by introducing args for your component's stories. It reduces the boilerplate code you'll need to write and maintain for each story.

Using the play function

* Storybook's play function and the [@storybook/addon-interactions](https://storybook.js.org/addons/@storybook/addon-interactions) are convenient helper methods to test component scenarios that otherwise require user intervention. They're small code snippets that execute once your story renders.
* Without the help of the play function and the @storybook/addon-interactions, you had to write your own stories and manually interact with the component to test out each use case scenario possible.

Using parameters

* Parameters are Storybook’s method of defining static metadata for stories. A story’s parameters can be used to provide configuration to various add-ons at the level of a story or group of stories.

Using decorators

* Decorators are a mechanism to wrap a component in arbitrary markup when rendering a story. Components are often created with assumptions about ‘where’ they render. Your styles might expect a theme or layout wrapper, or your UI might expect specific context or data providers.

Stories for two or more components

* When building design systems or component libraries, you may have two or more components created to work together. For instance, if you have a parent List component, it may require child ListItem components.

# Chapter 3: The Args in Stories

## Args object

* 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 add-ons 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 add-ons that affect args
* The args object can be defined at the [story](https://storybook.js.org/docs/react/writing-stories/args#story-args), [component](https://storybook.js.org/docs/react/writing-stories/args#component-args) and [global level](https://storybook.js.org/docs/react/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

* To define the args of a single story, use the args CSF story key:



* These args will only apply to the story for which they are attached, although you can [reuse](https://storybook.js.org/docs/react/writing-stories/build-pages-with-storybook#args-composition-for-presentational-screens) them via JavaScript object reuse:



## Component args

* You can also define args at the component level; they will apply to all the component's stories unless you overwrite them. To do so, use the args key on the default CSF export:



## Global args

* You can also define args at the global level; they will apply to every component's stories unless you overwrite them. To do so, export the args key in your preview.js:



## Args composition

* You can separate the arguments into a story to compose in other stories. Here's how you can combine args for multiple stories of the same component.



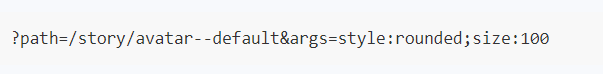
## Args can modify any aspect of your component

* You can use args in your stories to configure the component's appearance, similar to what you would do in an application. For example, here's how you could use a footer arg to populate a child component:



## Setting args through the URL

* You can also override the set of initial args for the active story by adding an args query parameter to the URL. Typically you would use the [Controls addon](https://storybook.js.org/docs/react/essentials/controls) to handle this. For example, here's how you could set a size and style arg in the Storybook's URL:



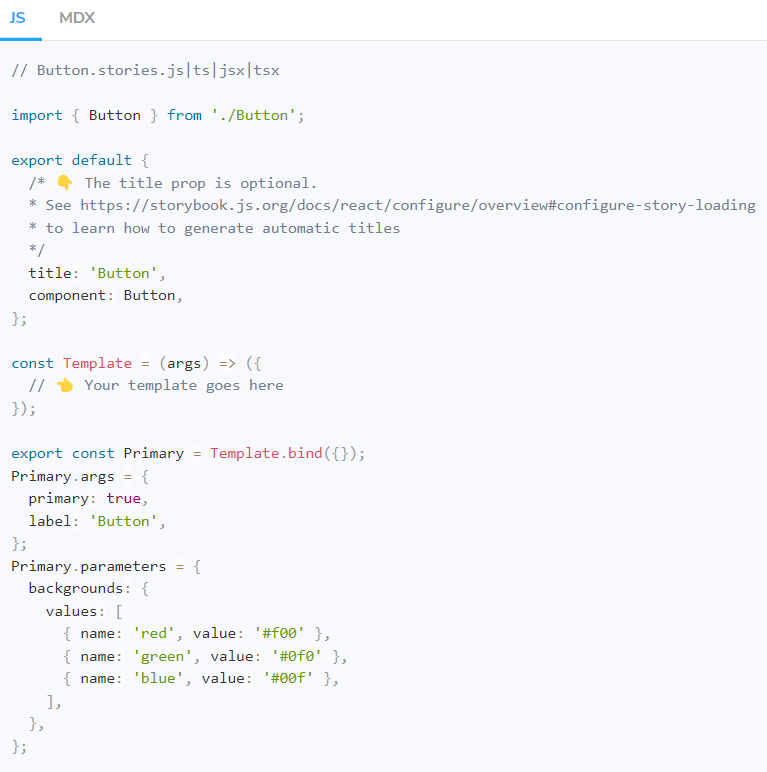
## Mapping to complex arg values

* Complex values such as JSX elements cannot be serialised to the manager (e.g., the Controls addon) or synced with the URL. Arg values can be "mapped" from a simple string to a complex type using the mapping property in argTypes to work around this limitation. It works in any arg but makes the most sense when used with the select control type.

**Chapter 4: The Parameters in Stories**

**Story parameters**

* Parameters are a set of static, named metadata about a story, typically used to control the behaviour of Storybook features and add-ons.
* For example, let’s customize the backgrounds addon via a parameter. We’ll use parameters.backgrounds to define which backgrounds appear in the backgrounds toolbar when a story is selected.
* We can set a parameter for a single story with the parameters key on a CSF export:



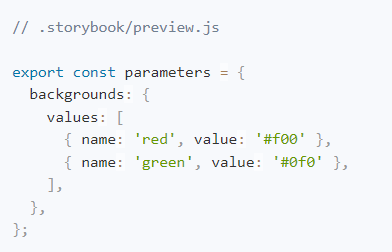
**Component parameters**

* We can set the parameters for all stories of a component using the parameters key on the default CSF export:



**Global parameters**

* We can also set the parameters for all stories via the parameters export of your [.storybook/preview.js](https://storybook.js.org/docs/react/configure/overview#configure-story-rendering) file (this is the file where you configure all stories):



**Rules of parameter inheritance**

* The way the global, component and story parameters are combined is:
  + More specific parameters take precedence (so a story parameter overwrites a component parameter which overwrites a global parameter).
  + Parameters are merged so keys are only ever overwritten, never dropped.
* The merging of parameters is important. It means it is possible to override a single specific sub-parameter on a per-story basis but still retain the majority of the parameters defined globally.

**Chapter 5: The Decorators in Stories**

**Wrap stories with extra markup**

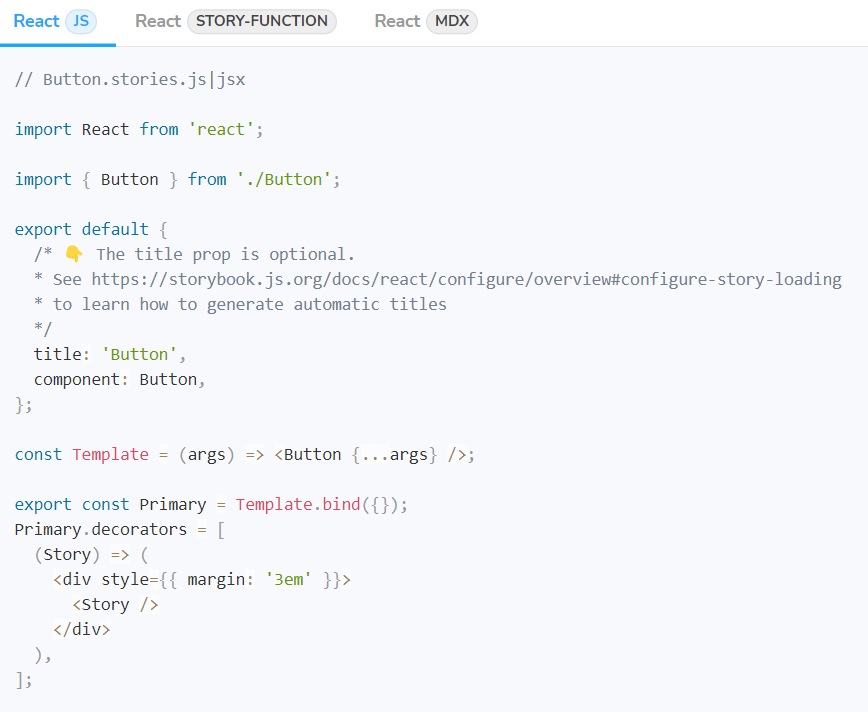
* A decorator is a way to wrap a story in extra “rendering” functionality. Many add-ons define decorators to augment your stories with extra rendering or gather details about how your story renders.
* When writing stories, decorators are typically used to wrap stories with extra markup or context mocking.
* Some components require a “harness” to render helpfully. For instance, if a component runs right up to its edges, you might want to space it inside Storybook. Use a decorator to add spacing for all stories of the component.

**“Context” for mocking**

* The second argument to a decorator function is the story context which in particular contains the keys:
  + args - the story arguments. You can use some [args](https://storybook.js.org/docs/react/writing-stories/args) in your decorators and drop them in the story implementation itself.
  + argTypes- Storybook's [argTypes](https://storybook.js.org/docs/react/api/argtypes) allow you to customize and fine-tune your stories [args](https://storybook.js.org/docs/react/writing-stories/args).
  + globals - Storybook-wide [globals](https://storybook.js.org/docs/react/essentials/toolbars-and-globals#globals). In particular, you can use the [toolbars feature](https://storybook.js.org/docs/react/essentials/toolbars-and-globals#global-types-toolbar-annotations) to allow you to change these values using Storybook’s UI.
  + hooks - Storybook's API hooks (e.g., useArgs).
  + parameters- the story's static metadata, most commonly used to control Storybook's behaviour of features and add-ons.
  + ViewModel- Storybook's current active window (e.g., canvas, docs).

**Story decorators**

* To define a decorator for a single story, use the decorators key on a named export



**Component decorators**

* To define a decorator for all stories of a component, use the decorators key of the default CSF export:



**Global decorators**

* We can also set a decorator for all stories via the decorators export of your [.storybook/preview.js](https://storybook.js.org/docs/react/configure/overview#configure-story-rendering) file (this is the file where you configure all stories):



**Decorator inheritance**

* Like parameters, decorators can be defined globally, at the component level, and for a single story (as we’ve seen).
* All decorators relevant to a story will run in the following order once the story renders:
  + Global decorators, in the order they are defined
  + Component decorators, in the order they are defined
  + Story decorators, in the order they are defined

**Chapter 6: The Play function in Stories**

**Setup the interactions addon**

* Play functions are small snippets of code executed after the story renders. Enabling you to interact with your components and test scenarios that otherwise required user intervention.
* We recommend installing Storybook's addon interactions before you start writing stories with the play function. It's the perfect complement, including a handy set of UI controls to allow you to command the execution flow. At any time, you can pause, resume, rewind, and step through each interaction. Also providing you with an easy-to-use debugger for potential issues.
* Run the following command to install the addon and the required dependencies.

npm install @storybook/testing-library @storybook/jest storybook/addon-interactions --save-dev

* Update your Storybook configuration (in .storybook/main.js) to include the interactions addon.



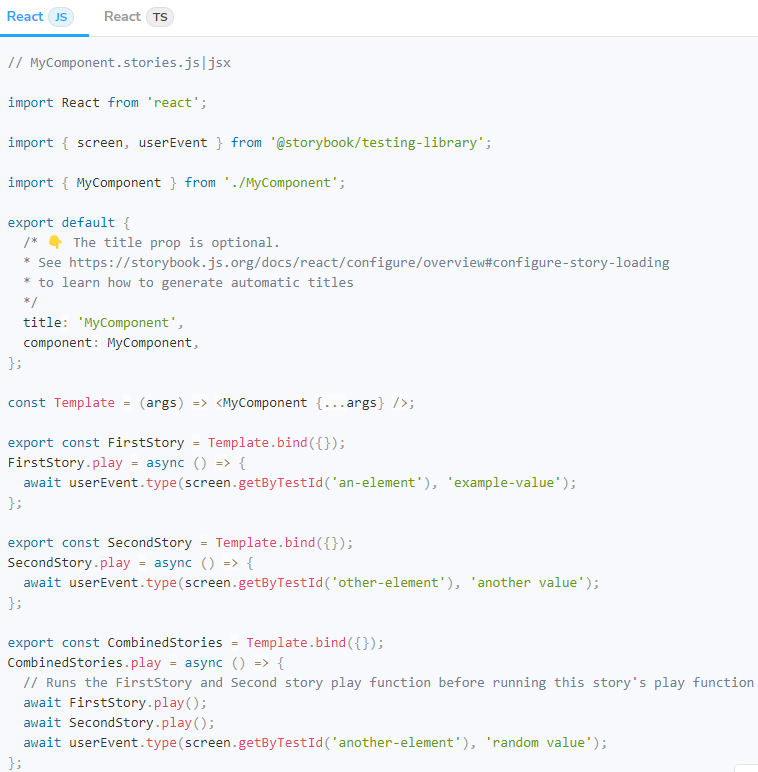
**Writing stories with the play function**

* Storybook's play functions are small code snippets that run once the story finishes rendering. Aided by the addon interactions, it allows you to build component interactions and test scenarios that were impossible without user intervention. For example, if you were working on a registration form and wanted to validate it, you could write the following story with the play function:



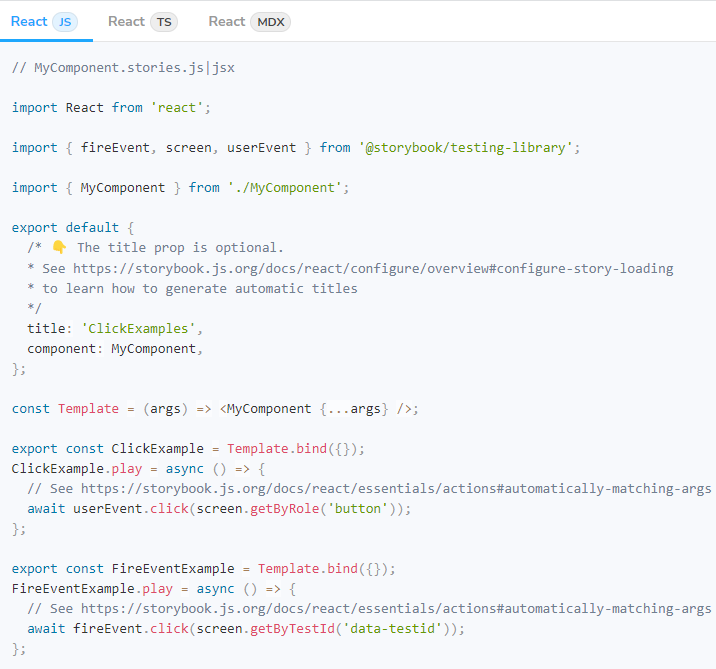
**Composing stories**

* You can also combine your play functions, similar to other existing Storybook features (e.g., [args](https://storybook.js.org/docs/react/writing-stories/args)). For example, if you wanted to verify a specific workflow for your component, you could write the following stories:



**Working with events**

* With the play function, you can incorporate the same level of interaction into your stories.
* A common type of component interaction is a button click. If you need to reproduce it in your story, you can define your story's play function as the following:



**Working with the Canvas**

* By default, each interaction you write inside your play function will be executed starting from the top-level element of the Canvas. This is acceptable for smaller components (e.g., buttons, checkboxes, text inputs), but can be inefficient for complex components (e.g., forms, pages), or for multiple stories. To accommodate this, you can adjust your interactions to start execution from the component's root. For example:

