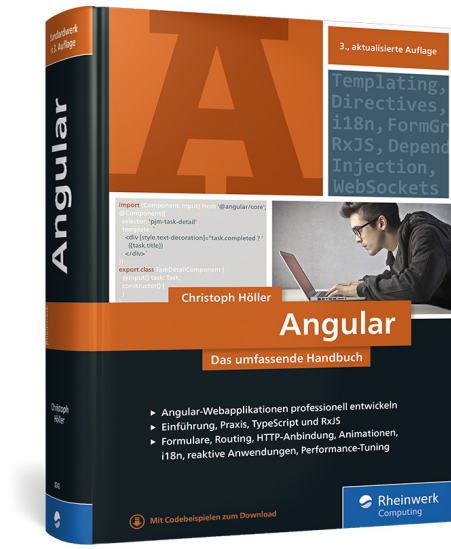
Christoph Höller

Storybook: Komponenten unabhängig von Ihrer Anwendung entwickeln und dokumentieren

Zusatzinformationen zu Angular



Rheinwerk Verlag GmbH, Bonn 2022
ISBN 978-3-8362-8243-7

Rheinwerk Computing

Bonuskapitel

Storybook: Komponenten unabhängig von Ihrer Anwendung entwickeln und dokumentieren

Die Implementierung von Komponenten unabhängig von einer laufenden Anwendung, verringert die Entwicklungszeit und sorgt ganz nebenbei für bessere APIs. Es gibt also direkt mehrere Gründe, sich Storybook einmal genauer anzuschauen!

Die Entwicklung von komplexen Komponenten innerhalb einer laufenden Anwendung kann oft mühsam (und fehleranfällig) sein. Die folgende Liste beschreibt nur einige der in diesem Kontext üblichen Probleme:

- Sie benötigen zunächst die vom Server geladenen Daten um Ihre Komponente zu rendern und müssen somit neben der Angular-Anwendung auch Ihr Backend starten, bevor Sie Ihre Komponente entwickeln können.
- Ihre Komponente hat eine besondere Darstellung für den Fall, dass keine Daten vorhanden sind, oder wenn ein Fehler beim Laden der Daten aufgetreten ist (Stichwort *Edge-Cases*).
- Um in die richtige Ansicht zu gelangen, müssen Sie zunächst einen Wizard durchlaufen, ein modales Fenster öffnen oder zum Ende der Seite scrollen.

Storybook versucht hier Abhilfe zu schaffen – anstatt Ihre Komponenten direkt in der laufenden Anwendung zu entwickeln, werden diese als sogenannte »Stories« in einer isolierten Umgebung zur Verfügung gestellt. Dabei haben Sie die Möglichkeit, beliebige Zustände der Komponente zu definieren und so z. B. auch die Darstellung von Edge-Cases immer im Blick zu haben.

Neben dem verbesserten Entwicklungs-Workflow bietet Ihnen das Vorgehen außerdem noch den großen Vorteil, dass Sie ganz nebenbei eine jederzeit aktuelle umfangreiche Dokumentation Ihrer Komponenten anlegen.

So werden Sie im Laufe dieses Kapitels lernen,

- was das *Component Story Format* (CSF) ist und wie Sie mit diesem Format Stories über Ihre Komponenten verfassen können.

- wie Sie Input-Bindings an Ihre Story weitergeben und diese sogar interaktiv über die Storybook-Oberfläche verändern können.
- wie Sie Output-Bindings testen können.
- was Sie bei der Arbeit mit Direktiven oder Komponenten die auf `ng-content` basieren beachten müssen.
- wie Sie unterschiedliche Auflösungen (Viewports) mit Storybook visualisieren können.
- was das *MDX*-Format ist und wie es Ihnen bei der Erstellung vollständiger Dokumentationen helfen kann.

Hinweis zu den Beispielquelltexten

Die Beispielquelltexte dieses Bonuskapitels können Sie unter *https://www.rheinwerk-verlag.de/5285* herunterladen. Sie finden Sie im Ordner *storybook*. Wie gewohnt erfolgt die Installation der Abhängigkeiten über `npm install`. Der Start von Storybook erfolgt anschließend über den Befehl `npm run storybook`.

Storybook in Ihr Projekt integrieren

Möchten Sie Storybook in Ihr Projekt integrieren, so müssen Sie hierfür lediglich den Befehl

```
npx sb init
```

im Wurzelverzeichnis Ihres Projekts ausführen. Der Installations-Wizard erkennt nun automatisch den Typ Ihres Projekts (*Angular*) und installiert die benötigten Abhängigkeiten. Zusätzlich werden einige benötigte Konfigurationsdateien, sowie eine Handvoll Beispiel-Stories in das Projekt integriert, sodass Sie Storybook nach erfolgreicher Installation über den Befehl

```
npm run storybook
```

starten können. Es sollte sich nun die in Abbildung 1 dargestellte Oberfläche öffnen.

Hinweis zur Aktualität von Storybook

An dieser Stelle möchte ich nicht unerwähnt lassen, dass der Einsatz von Storybook in der Praxis dazu führt, dass Sie Ihre Projekte in der Regel nicht direkt nach dem Release einer neuen Angular Version updaten können. So vergehen zwischen dem Release einer Angular-Major-Version und dem Release einer kompatiblen Storybook-Version in der Regel einige Wochen.

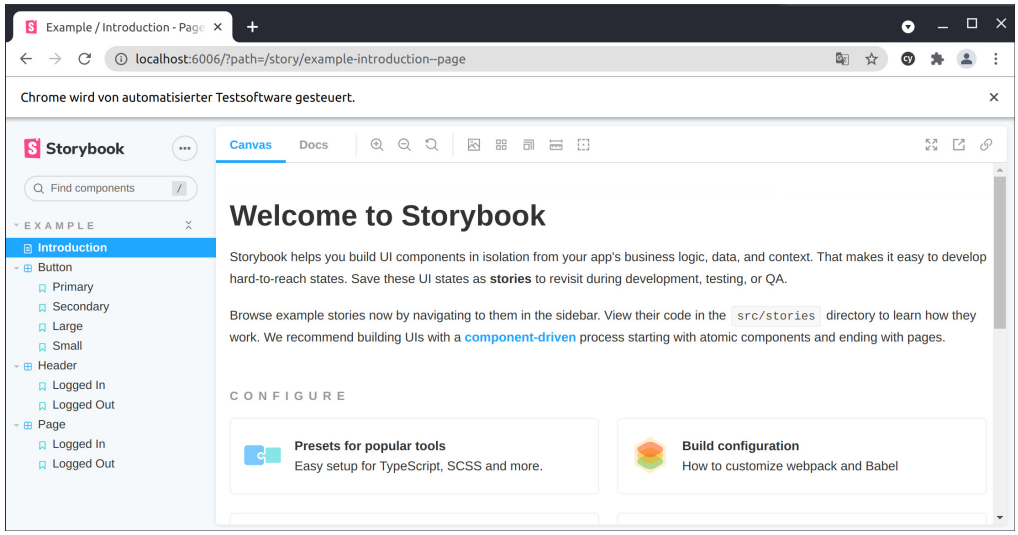


Abbildung 1 Der Willkommens-Bildschirm von Storybook

Das Component Story Format (CSF): Grundlagen von Storybook: Die erste Story

Die Beschreibung von Komponenten in *Storybook* erfolgt auf Basis des sogenannten *Component Story Format (CSF)*. In diesem Abschnitt möchte ich Ihnen die Grundlagen des Formats am Beispiel einer einfachen Kreis-Komponente (der `ColoredCircleComponent`) vorstellen. Die Komponente besitzt unter anderem `@Input`-Bindings für den Radius (`radius`), die Farbe der Füllung (`fill`) und ein Hintergrundbild (`imageUrl`).

Listing 1 zeigt die Implementierung der ersten Story für diese Komponente:

```
import {Meta, Story} from '@storybook/angular/types-6-0';
import {ColoredCircleComponent} from './colored-circle.component';

export default {
  title: 'Components/Colored Circle',
  component: ColoredCircleComponent,
} as Meta;

export const Simple: Story<ColoredCircleComponent> = (args:
ColoredCircleComponent) => ({
  props: {
    ...args,
```

```
    radius: 60,
  }
});
```

Listing 1 »colored-circle.stories.ts«: Die erste Story für die ColoredCircleComponent

Unerwartete Syntax des Component Story Format

Die Syntax des CSF wirkt auf Angular-Entwickler auf den ersten Blick oft etwas unerwartet. Der Grund hierfür liegt darin, dass Storybook ursprünglich für das Framework React entwickelt wurde und die Unterstützung von weiteren Frameworks wie Angular erst im Nachhinein hinzugefügt wurde. So orientieren sich beispielsweise die Begriffe für die Definition von Schnittstellen an der React-Syntax. Lassen Sie sich hiervon aber nicht abschrecken, ich werde Ihnen die Syntax in den nächsten Abschnitten Schritt für Schritt vorstellen.

Wie Sie sehen erfolgt die Definition von Stories in Dateien mit der Endung **.stories.ts*. Eine Story-Datei besteht dabei im Wesentlichen aus zwei Bestandteilen: Dem `default`-Export der die Komponente beschreibt sowie einem oder mehreren benannten (`named`) Exporten die die einzelnen Stories zu dieser Komponente beschreiben. So sorgen Sie über den Code

```
export default {
  title: 'Components/Colored Circle',
  component: ColoredCircleComponent,
} as Meta;
```

dafür, dass die Stories der `ColoredCircleComponent` in der Navigationsleiste von *Storybook* im Bereich *Components* mit dem Titel *Colored Circle* dargestellt werden.

Über den Export

```
export const Simple: Story<ColoredCircleComponent> = (args:
ColoredCircleComponent) => ({
  props: {
    ...args,
    radius: 100,
  }
});
```

definieren Sie anschließend die Story mit dem Titel »Simple« in Form einer *Arrow-Funktion*. Die Funktion erhält die Instanz der gerenderten Komponente als Eingangsparameter und erzeugt daraus ein `Story`-Objekt. Die `props`-Eigenschaft der Story ist dabei in etwa mit der Definition von `@Input`-Bindings in Angular vergleichbar. Über den Ausdruck

```
props: {
  ...args,
  radius: 100,
}
```

weisen Sie der `prop`-Eigenschaft der Story somit zunächst alle Werte der Komponenten-Instanz zu. Der Wert für die `radius`-Eigenschaft wird zusätzlich mit dem Wert `100` überschrieben.

Öffnen Sie nun erneut die *Storybook*-Oberfläche, so sollten Sie in der linken Navigationsleiste den Bereich COMPONENTS mit dem Unterpunkt COLORED CIRCLE sehen. ein Klick auf den Menü-Punkt sollte die in Abbildung 2 dargestellte Ansicht zeigen.

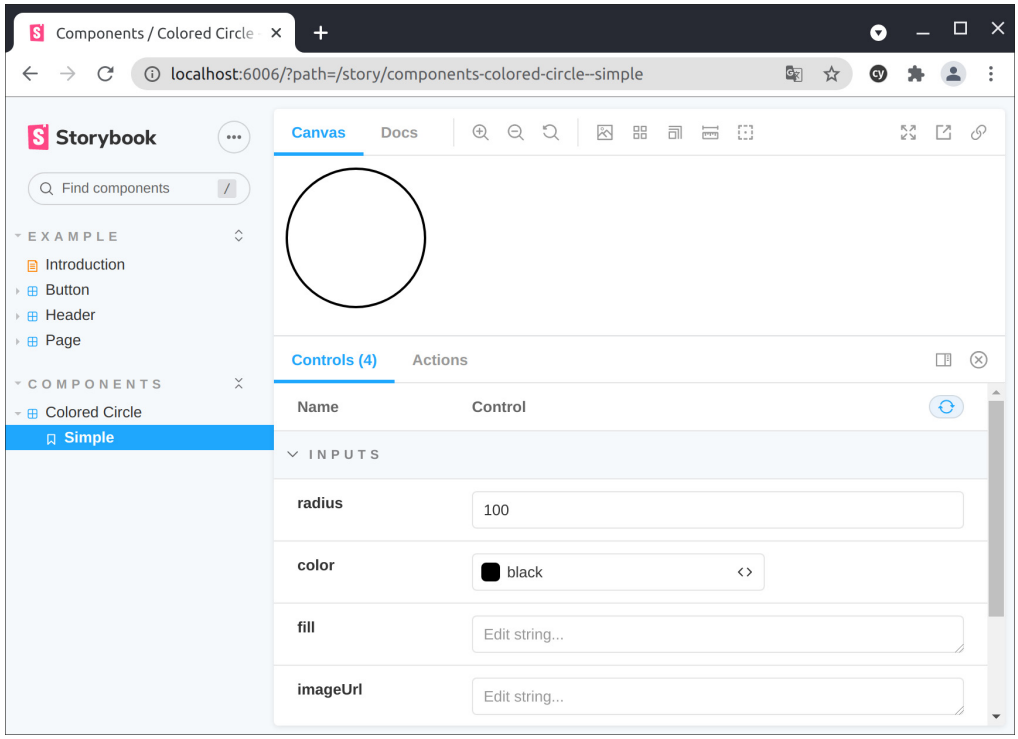


Abbildung 2 Ihre erste Story: Die Komponente »Colored Circle« in Storybook

Templates: Basis-Werte für Stories wiederverwenden

Großartig! Auf Basis dieser ersten Story, können Sie nun damit beginnen, Stories für beliebige weitere Darstellungen der Kreis-Komponente zu schreiben. Möchten Sie beispielsweise eine eigene Story für das Setzen der Hintergrundfarbe definieren, so können Sie dies wie folgt erreichen:

```
export const Simple: Story<ColoredCircleComponent> = (args:
ColoredCircleComponent) => ({
  props: {
    ...args,
    radius: 100,
  }
});

export const Filled: Story<ColoredCircleComponent> = (args:
ColoredCircleComponent) => ({
  props: {
    ...args,
    radius: 100,
    fill: 'red'
  }
});
```

Listing 2 »colored-circle.stories.ts«: Definition einer zweiten Story

In der *Storybook*-Oberfläche finden Sie neben der Simple-Story nun zusätzlich die *Filled*-Story (Abbildung 3).

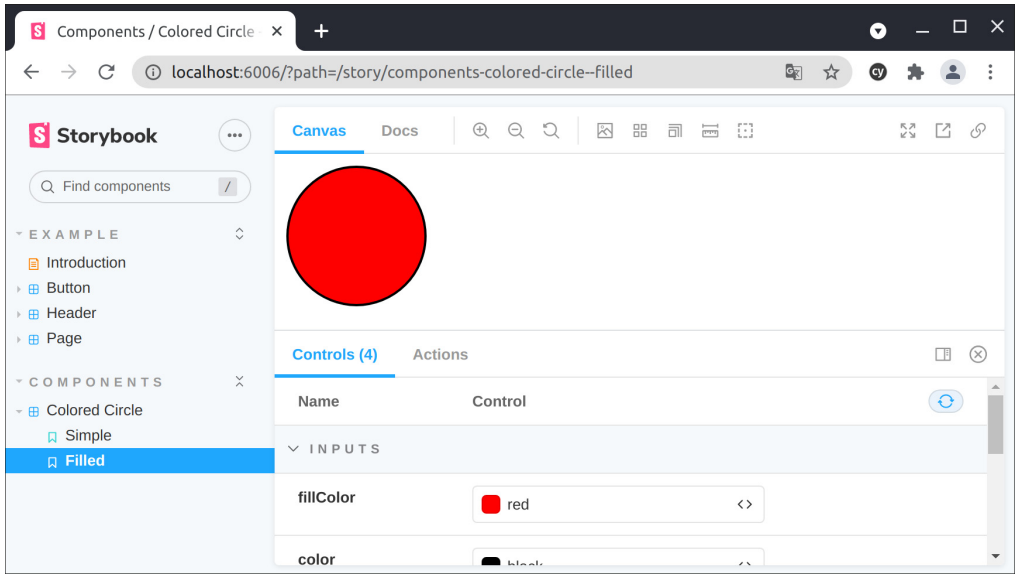


Abbildung 3 Ihre zweite Story: Der rot gefüllte Kreis

Wie Sie im Listing sehen, enthält der Code aber bereits jetzt viele Code-Duplikate. In diesem Fall bietet sich die Verwendung von *Template-Stories* an (siehe Listing 3).

```
export default {
  title: 'Components/Colored Circle',
  component: ColoredCircleComponent,
  args: {
    radius: 100
  }
} as Meta;

const Template : Story<ColoredCircleComponent> = (args:
ColoredCircleComponent) => ({
  props: args
});

export const Simple = Template.bind({})

export const Filled = Template.bind({});
Filled.args = {
  fill: 'red'
}

export const WithImage = Template.bind({});
WithImage.args = {
  imageUrl: 'http://lorempixel.com/200/200/cats'
}
```

Listing 3 »colored-circle.stories.ts«: Verwendung von Templates zur einfacheren Definition von Stories

Wie Sie sehen, haben Sie zunächst einmal die Möglichkeit gemeinsam genutzte Werte direkt im `default`-Export festzulegen:

```
export default {
  ...
  args: {
    radius: 100
  }
} as Meta;
```

Über den Ausdruck

```
const Template : Story<ColoredCircleComponent> = (args:
ColoredCircleComponent) => ({
  props: args
});
```

erstellen Sie anschließend eine Story, die Sie im weiteren Verlauf als Basis für die konkreten (exportierten) Stories verwenden können. Beachten Sie an dieser Stelle besonders die Zuweisung von Eigenschaften an eine generierte Story:

```
export const WithImage = Template.bind({});
WithImage.args = {
  imageUrl: 'http://lorempixel.com/200/200/cats'
}
```

Über die `bind`-Funktion wird zunächst eine Kopie der `Template`-Story angelegt. Die Zuweisung von Eigenschaften erfolgt hier über die `args`-Eigenschaft der Story-Instanz (und nicht wie man vermuten könnte über die `props`-Eigenschaft).

Ein Blick in die Oberfläche zeigt, dass Sie nun mit wenigen Zeilen Code drei unterschiedliche Stories für die `ColoredCircleComponent` verfasst haben (Abbildung 4).

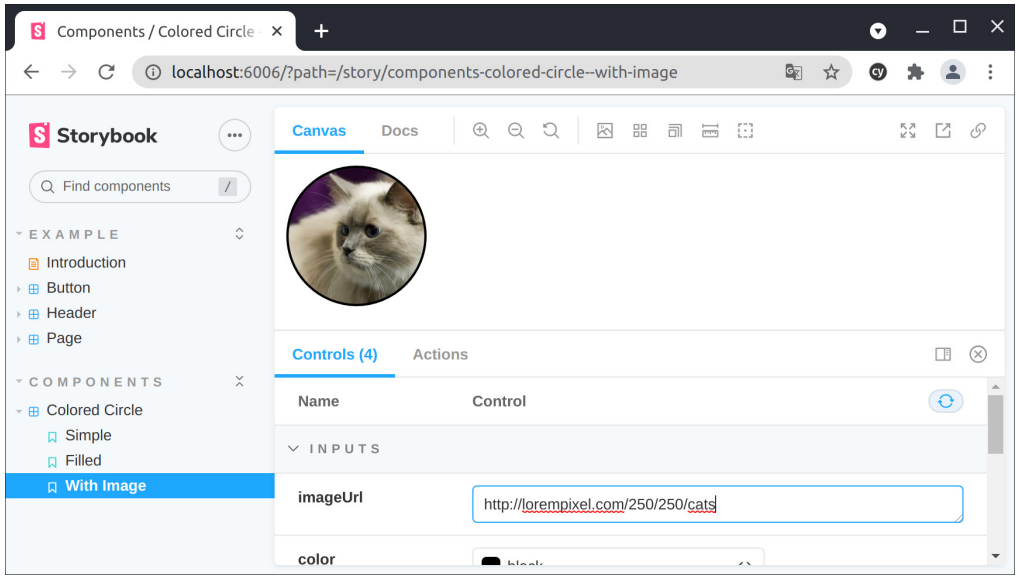


Abbildung 4 Verwendung von Templates zur einfacheren Definition von Stories

Templates ermöglichen es Ihnen so auf einfache Art und Weise *unterschiedliche Varianten* Ihrer Komponente vorzustellen.

Controls: Komponenten in der Story konfigurierbar machen

Ein weiteres sehr mächtiges Feature von *Storybook*, besteht in der Möglichkeit Input-Bindings interaktiv zu verändern. Das *Storybook* Standard-Setup ist dabei bereits so konfiguriert, dass unter jeder Story out-of-the-box ein Panel mit *Controls* dargestellt

wird, die zur Veränderung der Input-Bindings verwendet werden können. *Storybook* versucht an dieser Stelle auf Basis des Datentyps und des Namens eines Input-Bindings auf den richtigen Typ zu schließen. Wie Sie in Abbildung 5 sehen ist dies jedoch nicht immer automatisch möglich.

Abbildung 5 Das Controls-Feld ohne weitere Konfiguration

Die Input-Bindings `radius` und `color` wurden hier einmal auf Basis des Datentyps (`number`) und einmal auf Basis des Namens (`color`) automatisch korrekt erkannt. Bei der `fill`-Eigenschaft, die ebenfalls eine Farbe als Wert erhalten soll ist dies aber nicht passiert. In diesem Fall können Sie *Storybook* den Typ eines Feldes explizit über die `argTypes`-Eigenschaft des Default-Export mitteilen.

Listing 4 zeigt die Definition der `fill`-Eigenschaft als Farbe. Des Weiteren wird für die `imageUrl` eine Select-Liste mit vordefinierten Werten definiert.

```
export default {
  title: 'Components/Colored Circle',
  component: ColoredCircleComponent,
  ...
  argTypes: {
    fill: {
      name: 'Fill Color (fill)',
      description: 'The color that the circle is filled with ',
      control: 'color'
    },
    imageUrl: {
      options: [
        'http://lorempixel.com/200/200/cats',
        ...
      ],
      control: 'select'
```

```
    }
  }
} as Meta;
```

Listing 4 »colored-circle.stories.ts«: Festlegen der Controls für die einzelnen Input-Bindings

Beachten Sie an dieser Stelle, dass Sie *Storybook* nach dem Ändern der `argTypes`-Eigenschaft einmal komplett neu starten müssen. Ein anschließender Blick in die Oberfläche zeigt, dass beide Eigenschaften nun mit den vorgesehenen Controls ausgestattet sind.

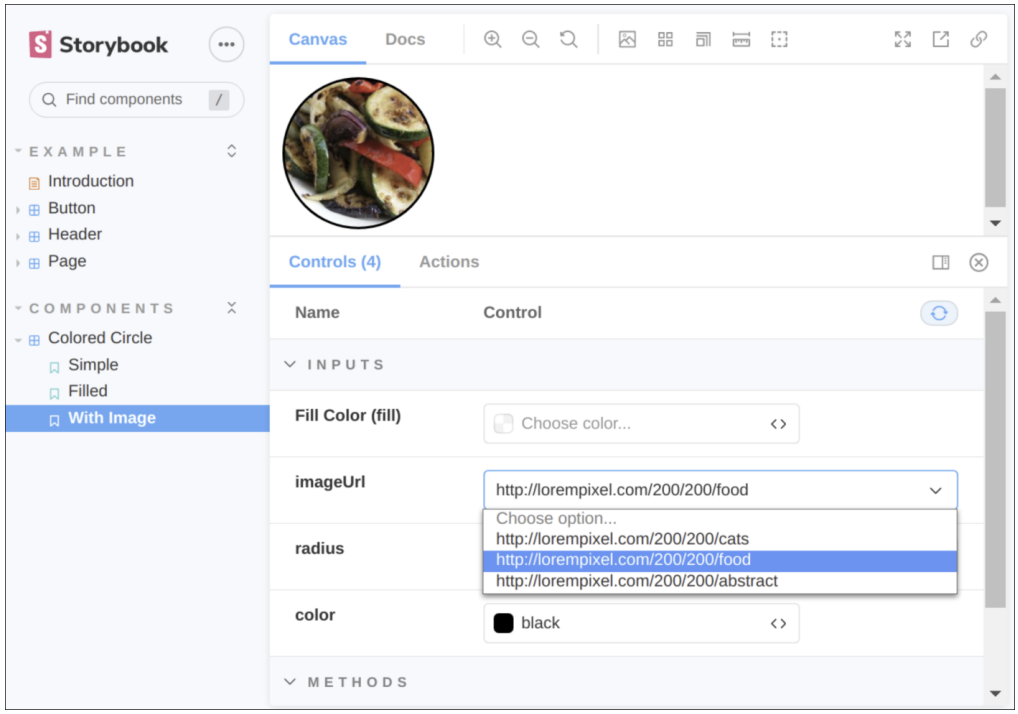


Abbildung 6 Select-Box für die Eigenschaft »imageUrl«

Storybook Actions: Output-Bindings visualisieren

Neben dem Setzen von Input-Bindings erlaubt *Storybook* es ebenfalls, die Funktionsweise von Output-Bindings zu visualisieren. Die Ausgabe von Werten, die über ein Output-Binding emitted werden, erfolgt dabei im Actions-Reiter des Control-Panels. Komfortabler Weise müssen Sie bei der Verwendung von Angular nichts weiter tun, um die Werte von Output-Bindings zu visualisieren: *Storybook* erkennt automatisch

alle Eigenschaften, die den `@Output`-Decorator besitzen, und gibt die emitteten Werte auf der Oberfläche aus. Abbildung 7 zeigt die Ausgabe des `clicked`-Output-Bindings, das die Position eines Klicks im Kreis ausgibt.

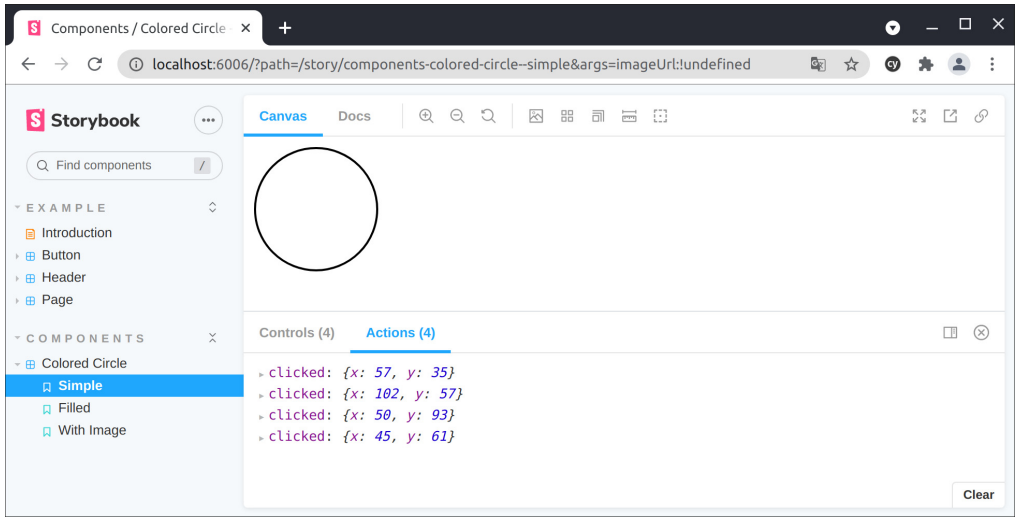


Abbildung 7 Ausgabe des »clicked«-Output-Bindings auf der Storybook Oberfläche

Explizite Definition von Templates ng-content & Direktiven rendern

Sie wissen nun, wie sie »einfache« Komponenten wie die `ColoredCircleComponent` mit *Storybook* entwickeln und visualisieren können. Doch was ist mit *Direktiven* (die ja kein eigenes Template besitzen) oder mit Komponenten die Ihren eigentlichen Inhalt über `ng-content` empfangen? *Storybook* erlaubt es für diese Fälle, das gerenderte Template explizit zu definieren. Schauen Sie sich für ein besseres Verständnis dieser Problemstellung zunächst einmal die `PanelComponent` aus Abbildung 8 an:

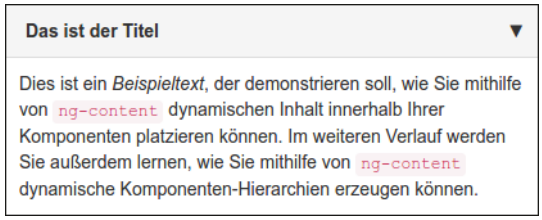


Abbildung 8 Ausklappbare Panel-Komponente zur Darstellung von dynamischem Inhalt

Die Komponente verwendet zur Definition des dargestellten Inhalts den `ng-content` Mechanismus (siehe Listing 5).

```
<ch-panel title="Das ist der Titel">
  Dies ist ein <i>Beispieltext</i>, der demonstrieren soll,
  wie Sie mithilfe von <code>ng-content</code> dynamischen Inhalt
  innerhalb Ihrer Komponenten platzieren können...
</ch-panel>
```

Listing 5 »panel-demo.component.ts«: Verwendung der »PanelComponent«

Sollten Sie mein Buch – Angular das umfassende Handbuch – aus dem Rheinwerk Verlag besitzen, können Sie sämtliche Details zum Thema Content-Insertion mittels `ng-content` in Abschnitt 3.6, »Content-Insertion: dynamische Komponentenhierarchien erstellen« nachlesen. Für den Moment reicht es aber zu verstehen, dass Sie für die Verwendung der Komponente die Möglichkeit benötigen, das umliegende Template zu definieren.

Möchten Sie nun also Stories für die `PanelComponent` erstellen, so können Sie dieses Template, wie in Listing 6 dargestellt, an die Story übergeben.

```
export default {
  title: 'Components/Panel Component',
  component: PanelComponent,
} as Meta;

export const Simple =  (args: PanelComponent) => ({
  template: `<ch-panel title="Abschnitt 2">
    Lorem ipsum dolor sit amet, consetetur...
  </ch-panel>`,
  props: args,
});
```

Listing 6 Definition der Panel Story mit eigenem Template

Wie in Abbildung 9 zu sehen ist, steht Ihnen auch hier wieder ein Control-Panel zur Verfügung, über das Sie die Input-Bindings, wie z. B. den Titel des Panels, dynamisch verändern können.

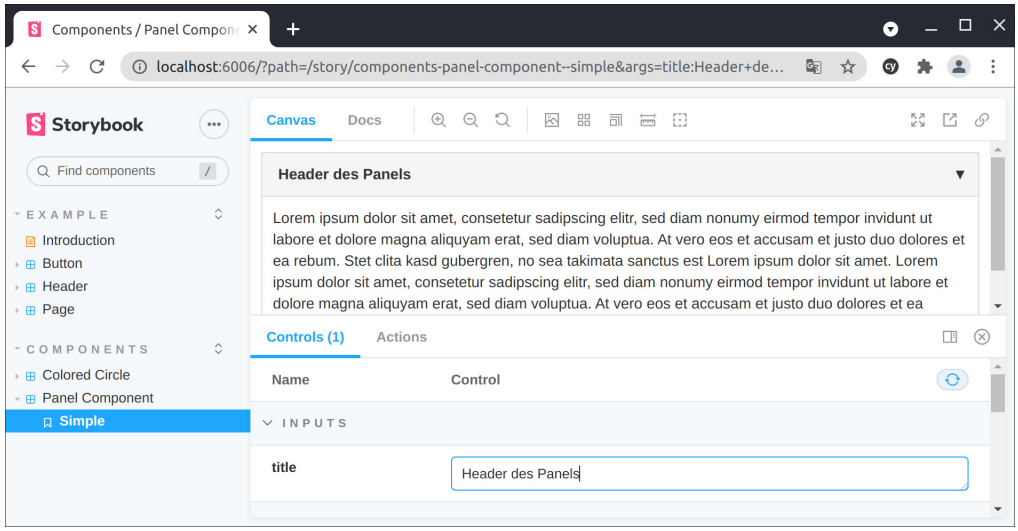


Abbildung 9 Darstellung der Panel-Komponente in Storybook

Module-Metadata: Erweiterte Konfiguration der Angular-Umgebung

Im Kontext der `PanelComponent` stelle ich Ihnen in meinem Buch in Kapitel 4, »Direktiven: Komponenten ohne eigenes Template«, , die `AccordionDirective` vor. Die Direktive hat die Aufgabe mehrere `PanelComponent` Instanzen zu gruppieren. Wird ein Panel ausgeklappt werden automatisch alle anderen Panels eingeklappt (siehe Abbildung 10).

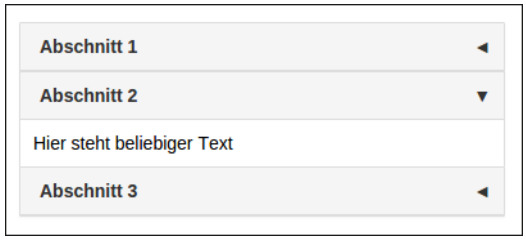


Abbildung 10 Einsatz der Accordion-Direktive zum gruppieren mehrere Panel-Komponenten

Listing 7 zeigt die Implementierung einer Story, die die Verwendung der Direktive vorstellt.

```
export const Accordion =  (args: PanelComponent) => ({
  template: `<div chAccoridon #firstAccordion="accordion" [onlyOneOpen]="true">
      <ch-panel title="Abschnitt 1">...</ch-panel>
      <ch-panel title="Abschnitt 2">Lorem ipsum dolor sit amet ...</ch-panel>
      <ch-panel title="Abschnitt 3">...</ch-panel>
    </div>`,
  props: args,
});
```

Listing 7 »panel.stories.ts«: Implementierung der Accordion-Story

Leider schlägt das Rendern dieser Story aber zunächst mit der in Abbildung 11 dargestellten Meldung fehl.

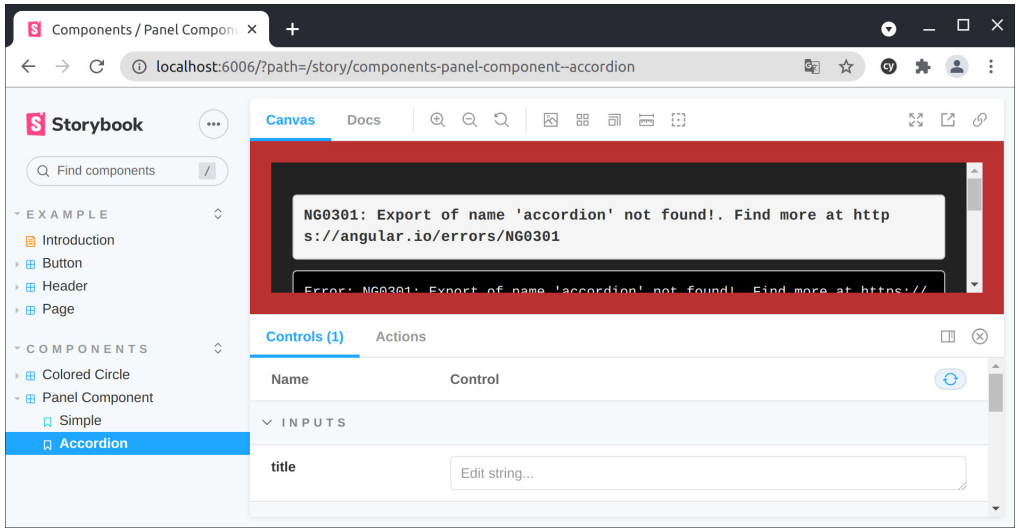


Abbildung 11 Fehlerhafte Story durch nicht gefundene Direktive

Das Problem besteht an dieser Stelle darin, dass Ihre Story bislang lediglich die `PanelComponent` kennt:

```
export default {
  title: 'Components/Panel Component',
  component: PanelComponent,
} as Meta;
```

Ohne weitere Konfiguration erzeugt *Storybook* nun unter der Haube ein `NgModule`, das lediglich die `PanelComponent` deklariert, und versucht damit die implementierten Stories zu rendern.

Möchten Sie nun zusätzlich zur `PanelComponent` die `AccordionDirective` verwenden, stellt Ihnen *Storybook* die Funktion `moduleMetadata` bereit, über die Sie manuell ein komplettes `NgModule` für die *Storybook*-Umgebung definieren können. Listing 8 und Listing 9 zeigen die Definition des `PanelModule` sowie die anschließende Integration des Moduls in Ihre *Storybook*-Story.

```
@NgModule({
  ...
  exports: [
   PanelComponent, PanelHeaderDirective, AccordionDirective
  ],
})
export class PanelModule {
}
```

Listing 8 »panel.module.ts«: »NgModule« für die Panel-Funktionalität

```
import {Meta} from '@storybook/angular/types-6-0';
import {PanelComponent} from './panel.component';
import {PanelModule} from './panel.module';
import {moduleMetadata} from '@storybook/angular';

export default {
  title: 'Components/Panel Component',
  component: PanelComponent,
  decorators: [moduleMetadata({
    imports: [
      PanelModule
    ]
  })],
} as Meta;
```

Listing 9 »panel.stories.ts«: Verwendung der »moduleMetadata«-Funktion zur manuellen Definition der Abhängigkeiten der Story

Neben dem Import von Modulen, stellt die `moduleMetadata`-Funktion Ihnen die Möglichkeit bereit sämtliche Eigenschaft die Sie ansonsten über den `@NgModule`-Decorator definiert hätten festzulegen. So könnten Sie beispielsweise über das `providers`-Array Services bereitstellen, die von Ihrer Komponente verwendet werden.

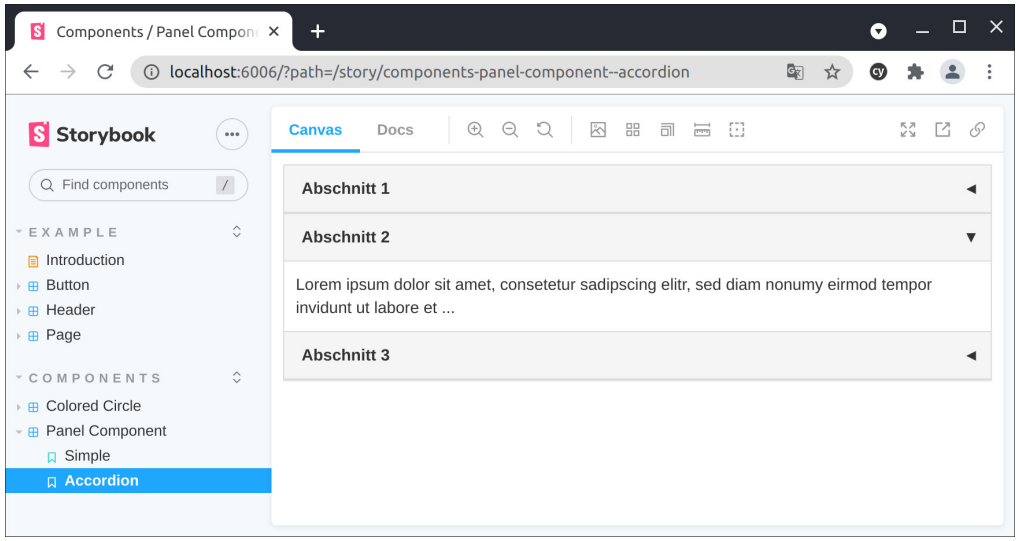


Abbildung 12 Darstellung des Accordion in Storybook

Viewport: Unterschiedliche Auflösungen testen

Durch die zunehmende Verbreitung mobiler Endgeräte ist der Anteil der mobilen Internetnutzung in den letzten Jahren konstant gestiegen und liegt aktuell bereits bei über 80 %. Umso wichtiger ist es, sicherzustellen, dass Ihre Anwendungen und Komponenten in unterschiedlichen Auflösungen (*Viewports*) gut bedienbar sind.

Storybook bietet Ihnen in diesem Zusammenhang die Möglichkeit den *Viewport*, in dem Ihre Story gerendert werden soll, über den in Abbildung 13 dargestellten Button zu verändern.

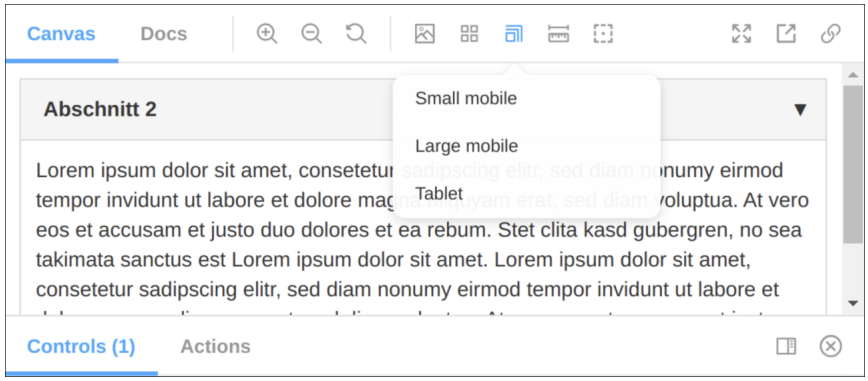


Abbildung 13 Auswahl des Viewports in der Storybook Oberfläche

Die Auswahl SMALL MOBILE sollte anschließend zu einer Darstellung wie in Abbildung 14 führen.

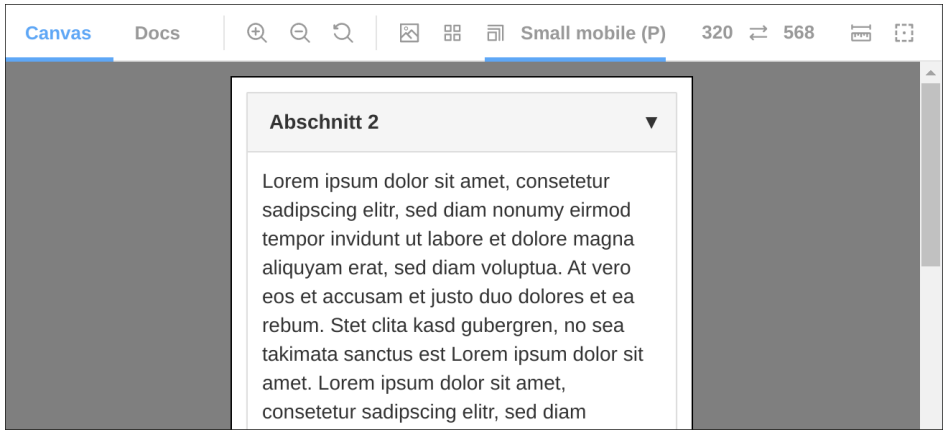


Abbildung 14 Mobile Darstellung der Panel-Komponente

Neue Viewports definieren

Ohne weitere Konfiguration stehen Ihnen an dieser Stelle die vier Optionen RESPONSIVE (der Standard-Wert), SMALL MOBILE, LARGE MOBILE und TABLET zur Verfügung. Möchten Sie diese Liste mit selbst definierten *Viewports* ergänzen, so können Sie dies über die zentrale *Storybook*-Konfigurationsdatei *preview.js* tun. Listing 10 zeigt die Erweiterung der Standard-Viewports um zwei weitere Einträge.

```
import {MINIMAL_VIEWPORTS} from "@storybook/addon-viewport";

const customViewports = {
  kindleFire2: {
    name: 'Kindle Fire 2',
    styles: {
      width: '600px',
      height: '963px',
    },
  },
  kindleFireHD: { ... },
};

export const parameters = {
  ...
  viewport: {
    viewports: {
      ...MINIMAL_VIEWPORTS,
```

```
      ...customViewports,
    },
  }
}
```

Listing 10 »storybook/preview.js«: Hinzufügen neuer Viewports

Die exportierte Eigenschaft `parameters` enthält an dieser Stelle diverse globale Einstellungen. Die Definition von eigenen *Viewports* geschieht dabei über die Eigenschaft `viewport`, die ihrerseits in der Eigenschaft `viewports` ein Objektliteral mit den zur Verfügung stehenden *Viewports* erhält. Über den *Spread-Operator* werden die Standard-Werte (`MINIMAL_VIEWPORTS`) mit den selbst definierten *Viewports* (`customViewports`) erweitert.

Nach einem Neustart von Storybook sollten Sie nun das in Abbildung 15 dargestellte Viewport-Menü sehen.

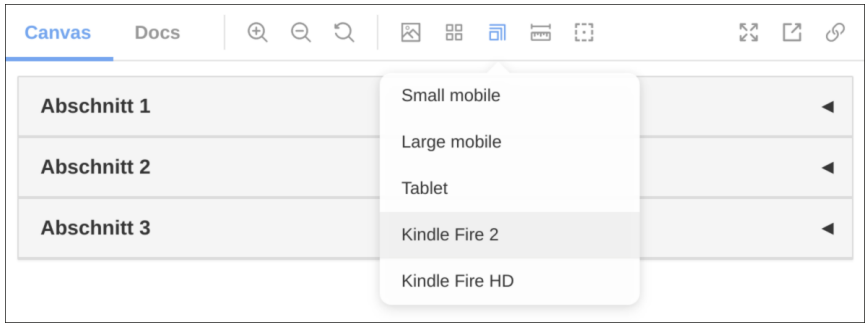


Abbildung 15 Erweiterte Viewport-Liste

INITIAL_VIEWPORTS: Die Gerätespezifische Viewport-Liste von Storybook verwenden

Neben dem Wert `MINIMAL_VIEWPORTS` stellt *Storybook* Ihnen out-of-the-box zusätzlich die Konstante `INITIAL_VIEWPORTS` zur Verfügung. Diese enthält zum aktuellen Zeitpunkt ca. zwanzig gerätespezifische Konfigurationen (etwa für das *Samsung Galaxy 6* oder das *Apple iPhone 12*). Möchten Sie Ihre Komponenten für eine Vielzahl von Geräten überprüfen, können Sie in der *preview.js* also einfach direkt diese Konstante mit der `viewports`-Eigenschaft verknüpfen:

```
viewports: {
  ...INITIAL_VIEWPORTS,
  ...customViewports,
},
```

Den Standard-Viewport einer Story festlegen

Neben der Möglichkeit den Viewport über das Menü manuell auszuwählen, können Sie des Weiteren pro Story oder pro Komponente bestimmen, mit welchem Standard-Viewport diese dargestellt werden soll.

Die Konfigurations-Eigenschaft der Story erwartet dabei den Schlüssel aus dem `viewports`-Objektliteral. Leider ist dieser Schlüssel in der Storybook-Oberfläche aber nicht ersichtlich, sodass Sie sich eines kleinen Tricks bedienen müssen: Geben Sie das Objektliteral einfach über die `console.log`-Funktion in der Developer-Konsole aus:

```
import {MINIMAL_VIEWPORTS} from "@storybook/addon-viewport";
...
console.log('ViewPorts', MINIMAL_VIEWPORTS)
```

Listing 11 »storybook/preview.js«: Ausgabe der vordefinierten »ViewPorts« auf der Developer Konsole

In Ihrem Browser sollten Sie nun die in Abbildung 16 dargestellte Ausgabe in der Konsole sehen.

```
ViewPorts ▼ {mobile1: {…}, mobile2: {…}, tablet: {…}}
            ▶ mobile1: {name: "Small mobile", styles: {…}, type: "mobile"}
            ▶ mobile2: {name: "Large mobile", styles: {…}, type: "mobile"}
            ▶ tablet: {name: "Tablet", styles: {…}, type: "tablet"}
            ▶ __proto__: Object
```

Abbildung 16 Ausgabe der vordefinierten »ViewPorts« auf der Developer Konsole

Möchten Sie nun für die `PanelComponent` eine zusätzliche Story verfassen, die Standard-mäßig den Viewport SMALL MOBILE verwendet, so können Sie dies wie in Listing 12 dargestellt erreichen.

```
export const MobileView: Story<PanelComponent> = (args: PanelComponent) => ({
  template: `<ch-panel title="Abschnitt 2">
    Lorem ipsum dolor sit amet, consetetur sadipscing...
  </ch-panel>`,
  props: args,
});

MobileView.parameters = {
  viewport: {
    defaultViewport: 'mobile1'
  },
};
```

Listing 12 »panel.stories.ts«: Festlegen des Standard-Viewports pro Story

In der Oberfläche wird die Story MOBILE VIEW nun wie erwartet mit geringerer Auflösung dargestellt (Abbildung 17).

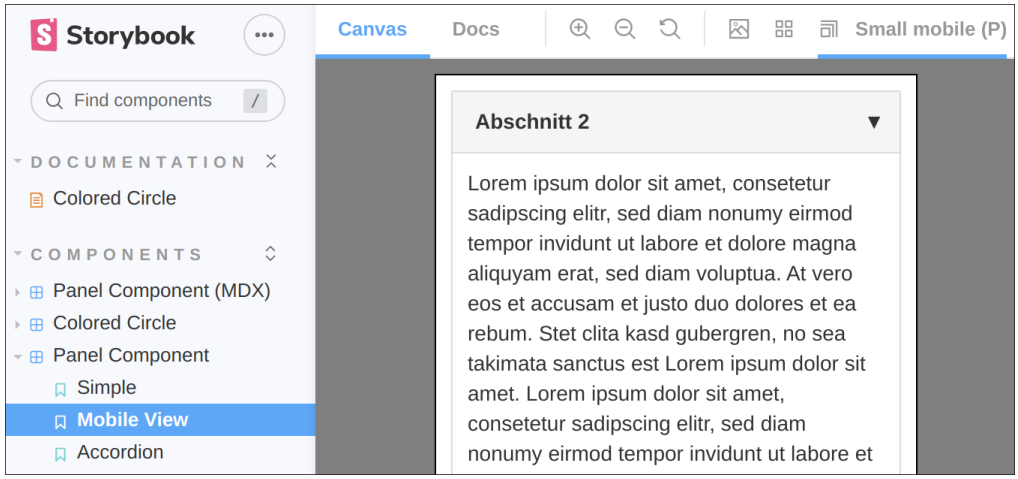


Abbildung 17 Story für die mobile Darstellung der Panel-Komponente

Docs & MDX: Vollständige Dokumentationen mit Storybook erstellen

Sie wissen nun, wie Sie *Storybook* dazu verwenden können Ihre Komponenten unabhängig von einer laufenden Anwendung zu entwickeln und mit unterschiedlichen Eigenschaften und Auflösungen darzustellen. Darüber hinaus bietet *Storybook* Ihnen über das *Docs-Plugin* aber zusätzlich die Möglichkeit, vollständige Dokumentationen Ihrer Komponenten, bzw. Ihrer gesamten Komponentenbibliothek zu verfassen.

Der Docs-Tab: Standard-Dokumentation von Storybook

Wie Sie vermutlich gesehen haben, erstellt *Storybook* bereits ohne weitere Konfiguration, zusätzlich zu den konkreten Stories, eine Dokumentation in der alle Stories zu einer Komponente zusammengefasst werden. Diese Dokumentation findet sich über den Reiter *Docs* im oberen Bereich einer Story (Abbildung 18).

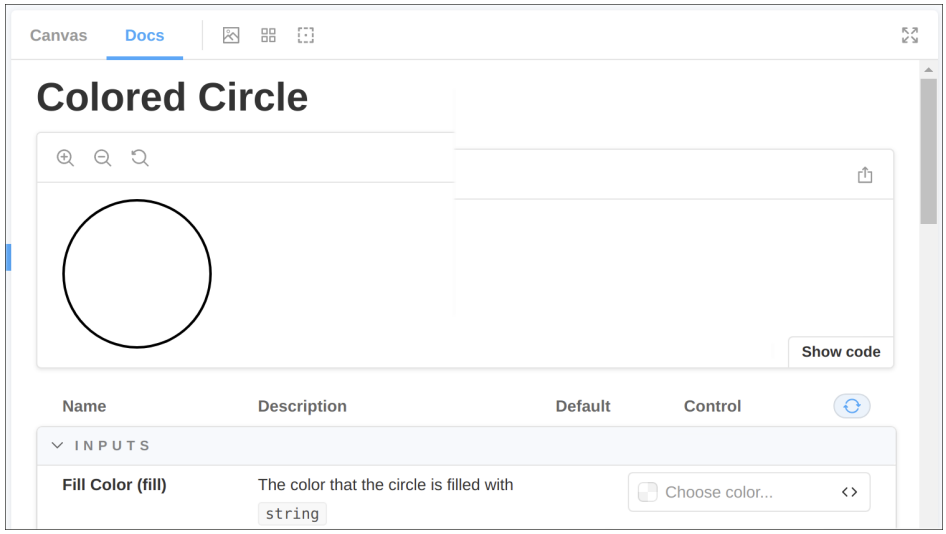


Abbildung 18 Der Docs-Tab der »ColoredCircle«-Story

Die Beschreibung von Komponenten und Stories verändern

Möchten Sie den Nutzern Ihrer Komponente lediglich einige zusätzliche Prosa-Informationen im Allgemeinen oder zur spezifischen Story mitgeben, so können Sie dies über die `parameters.docs`-Eigenschaft erreichen. Listing 13 zeigt die Definition der Komponenten-Beschreibung, sowie der Beschreibung der `WithImage`-Story im *CSF-Format*:

```
export default {
  title: 'Components/Colored Circle',
  component: ColoredCircleComponent,
  parameters: {
    docs: {
      description: {
        component: 'Die ` coloredCircleComponent` zeichnet sehr schöne Kreise'
      },
    }
  },
  ...
} as Meta;
...
export const WithImage = Template.bind({});
WithImage.parameters = {
  docs: {
    description: {
```

```
      story: 'Möchten Sie Ihrem Kreis in Hintergrund-Bild geben, ...
    },
  },
};
```

Listing 13 »colored-circle.stories.ts«: Hinzufügen einer Beschreibung für eine Story

Abbildung 19 und Abbildung 20 zeigen die jeweiligen Beschreibungen in der *Storybook*-Oberfläche.

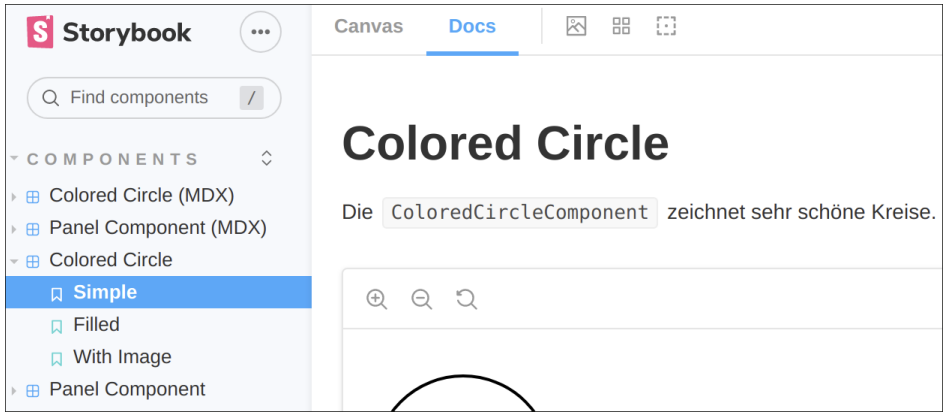


Abbildung 19 Darstellung der Beschreibung der Komponente im Docs-Tab

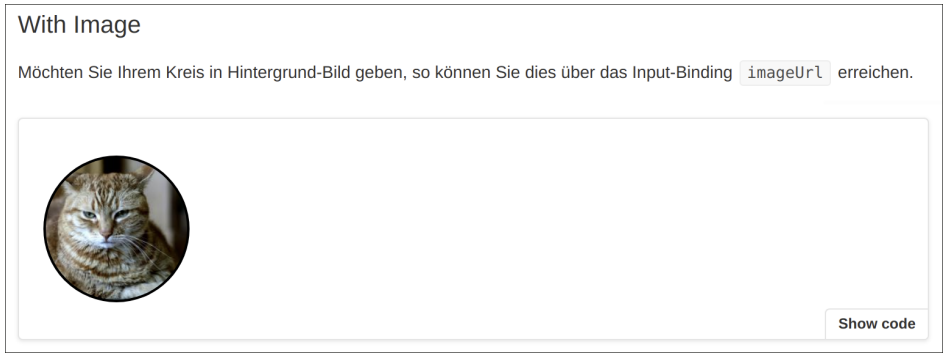


Abbildung 20 Darstellung der Beschreibung der Story im Docs-Tab

MDX: Eigene Dokumentationen schreiben und in Storybook integrieren

Neben der Möglichkeit, Dokumentationen automatisch generieren zu lassen, ist es außerdem sehr leicht möglich eigene Dokumentationen in *Storybook* zu verfassen. Das Framework stellt Ihnen für diesen Anwendungsfall die propiertäre Syntax *MDX* zu Verfügung. So handelt es sich bei *MDX* um eine Mischung aus *Markdown (MD)* und *JSX* (der Rendering-Syntax von *React*).

Listing 14 zeigt die erste Implementierung einer *Introduction*-Seite für das Projekt.

```
import {Meta} from '@storybook/addon-docs';
<Meta
    title="Documentation/Introduction"
/>
# Sprachkern
In diesem Beispielprojekt lernen Sie die Kernkonzepte von Angular kennen.
Die folgende Liste gibt einen Überblick über die vorgestellten Themen:

## Input / Output Bindings
Hier lernen Sie, am Beispiel der `ColoredCircleComponent` wie Sie
Schnittstellen zu Komponenten definieren
```

Listing 14 »introduction.stories.mdx«: Implementierung der Projekt

Wie im *CSF-Format* wird die Information darüber, wo die Seite in der Navigation »aufgehängen«, über ein `Meta`-Objekt gesteuert. Im Fall von *MDX* erfolgt die Definition aber mit Hilfe einer sogenannten *JSX-Komponente* (der `Meta`-Komponente) Der Block

```
import {Meta} from '@storybook/addon-docs';
<Meta
    title="Documentation/Introduction"
/>
```

sorgt somit dafür, dass in der Navigation ein Bereich DOCUMENTATION mit der Unterseite INTRODUCTION erscheint.

Der eigentliche Inhalt der Dokumentation wird anschließend mit Hilfe der *Markdown*-Syntax verfasst, sodass Sie in der Oberfläche die in Abbildung 21 dargestellte Ansicht sehen sollten.

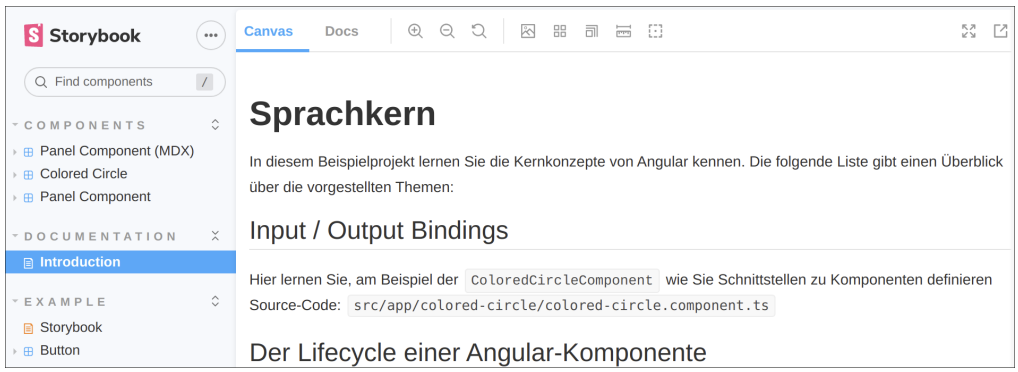


Abbildung 21 Einstiegsseite für die Dokumentation des Sprachkern-Projekts

Stylings & Bilder in MDX-Dokumente integrieren

Bislang sieht Ihre Dokumentation noch recht »trocken« aus. Glücklicherweise haben Sie in *MDX* aber sehr einfach die Möglichkeit, CSS-Styles zu definieren und Bilder in Ihre Dokumentation zu integrieren. Listing 15 zeigt exemplarisch das Hinzufügen des Angular-Logos aus dem `assets` -Ordner in Ihre Dokumentation.

```
import Logo from '../assets/angular-logo.png';
<Meta
    title="Documentation/Introduction"
/>
<style>{`
.logo {
 height: 60px;
 width: 60px;
}
`}</style>
<div className="logo">
    <img src={Logo} />
</div>
# Sprachkern
In diesem Beispielprojekt lernen Sie ...
```

Listing 15 »introduction.stories.mdx«: Styling und Import von Bildern in MDX

Beachten Sie, dass die Definition des Markups an dieser Stelle In *JSX*-Syntax erfolgt, so dass Sie dort dynamisch auf importierte Werte zugreifen können. Der Ausdruck

```
<img src={Logo} />
```

ist dabei vergleichbar mit der Verwendung von Input-Bindings in Angular. Abbildung 22 zeigt die neue Version Ihrer Dokumentation inklusive Logo.

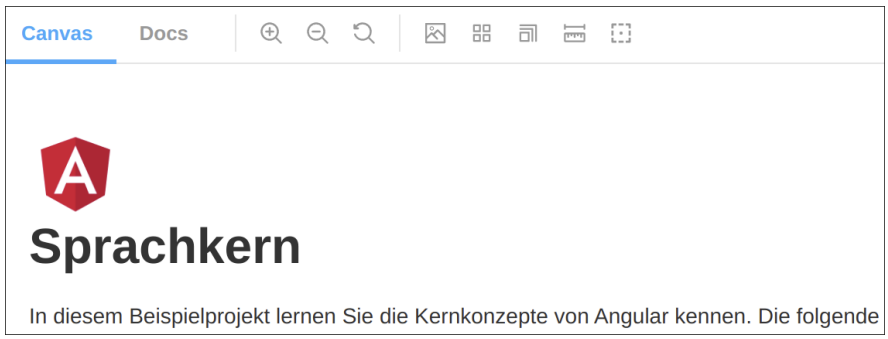


Abbildung 22 Styling und Import von Bildern in MDX

Zusätzliche Markdown-Dokumente importieren

In vielen Fällen enthält Ihr Source-Code ohnehin schon Dokumentation in Form von *Markdown*-Dateien. Typische Beispiele wären hier die obligatorische *README.md* oder der Versionsverlauf Ihres Projekts in der Datei *CHANGELOG.md*. Möchten Sie diese Dateien in Ihre *Storybook*-Dokumentation integrieren, so können Sie hierfür die *JSX*-Komponente `Description` verwenden (siehe Listing 16).

```
import {Meta, Description} from '@storybook/addon-docs';
import Changelog from '../../CHANGELOG.md';
...
# Sprachkern
...
<Description markdown={Changelog} />
```

Listing 16 »introduction.stories.mdx«: Integration der »CHANGELOG.md« in die Story

Wie bei der Integration des Logos wird der Inhalt der Datei *CHANGELOG.md* über eine `import`-Anweisung importiert. Das Rendern des *Markdown*-Codes erfolgt anschließend mit Hilfe der `Description`-Komponente:

```
<Description markdown={Changelog} />;
```

Ein Blick in die Oberfläche zeigt, dass Ihr Versionsverlauf nun Teil der Dokumentation ist (Abbildung 23).

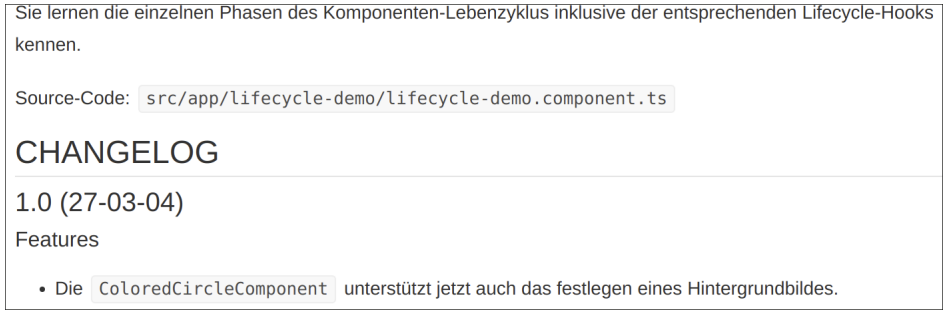


Abbildung 23 Integration der »CHANGELOG.md« in die Story

Stories in MDX einbetten

Ein weiteres interessantes Feature besteht in der Möglichkeit, bestehende Stories in Ihre *MDX*-Dokumente zu integrieren. Der Integration einer Story erfolgt dabei über die *Story-ID*. Diese finden Sie beim Öffnen einer Story in der Browser-URL (siehe Abbildung 24).

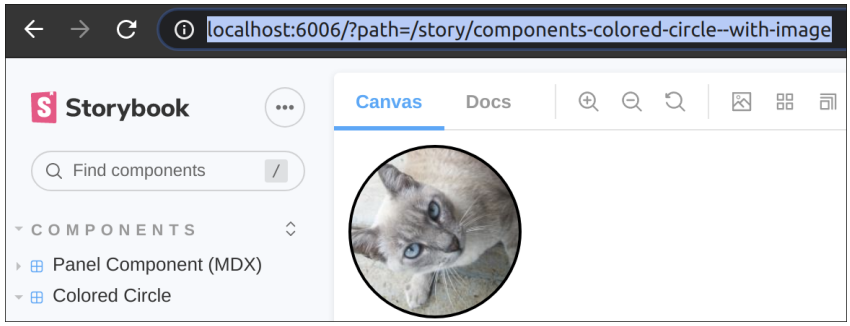


Abbildung 24 Die ID der Story in der Browser-URL

Möchten Sie die ausgewählte Story zu in Ihrer *MDX*-Dokumentation darstellen, so erfolgt dies über die `Story`-Komponente (siehe Listing 17).

```
import {Meta, Description, Story} from '@storybook/addon-docs';
...
## Input / Output Bindings
Hier lernen Sie, am Beispiel der `ColoredCircleComponent` ...
### Vorschau:
<Story id="components-colored-circle--with-image"/>
```

Listing 17 Einbetten der Story in die MDX-Dokumentation

Storybook rendert die Story nun innerhalb der *Introduction*-Seite (Abbildung 25).

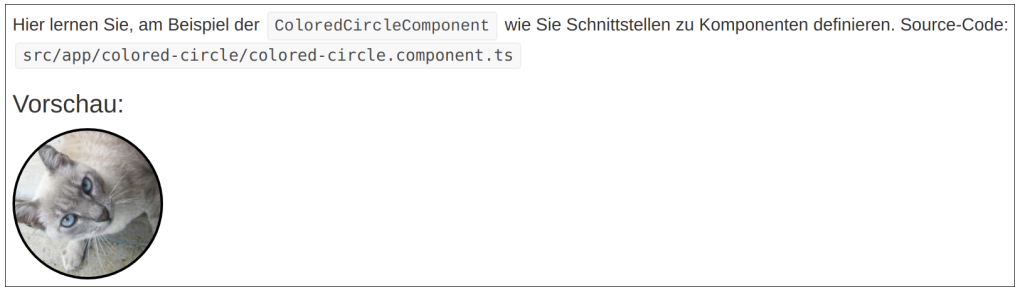


Abbildung 25 Die eingebettet Story in der MDX-Dokumentation

Komplette Stories in MDX verfassen

Bis hierhin haben Sie *MDX* lediglich dazu verwendet Ihre im *Component Story Format* verfassten Stories um Prosa-Dokumentation zu ergänzen. Darüber hinaus ist es aber ebenfalls möglich *MDX* als vollwertigen Ersatz für das CSF zu verwenden. So bieten Ihnen die bereits in den vorherigen Abschnitten vorgestellten *JSX*-Komponenten `Meta` und `Story` die Möglichkeit sämtliche Eigenschaften die zur Definition von Sto-

ries notwendig sind direkt in *MDX* zu verfassen. Listing 18 zeigt exemplarisch die Implementierung der `ColoredCircleComponent`-Stories im MDX-Format.

```
import {Meta, Story} from '@storybook/addon-docs/blocks';
import {ColoredCircleComponent} from './colored-circle.component';

<Meta
    title="Components/Colored Circle (MDX)"
    component={ColoredCircleComponent}
    args={{
        radius: 60
    }}
    argTypes={{
        fill: {
            name: 'Fill Color (fill)',
            description: 'The color that the circle is filled with ',
            control: 'color'
        },
    }}/>
export const Template = (args) => <ColoredCircleComponent {...args} />

# Einführung
Die `ColoredCircleComponent` ermöglicht Ihnen das Zeichen von Kreisen auf
Ihrer Oberfläche.
...
<Story name="Default">
    {Template.bind({})}
</Story>
```

Listing 18 »colored-circle.stories.mdx«: Definition kompletter Stories im MDX-Format

Wie Sie sehen, werden Konfigurations-Parameter wie `component`, `args` oder `argTypes` in Form von *JSX*-Bindings an die Meta-Komponente übergeben. Beachten Sie an dieser Stelle insbesondere die Verwendung von doppelten geschweiften Klammern, bei der Übergabe von Objektliteralen:

```
args={{
  radius: 60
}}
```

So leiten die äußeren geschweiften Klammern das *JSX*-Binding ein. Die inneren geschweiften Klammern definieren anschließend das übergebene Objektliteral.

Syntaxfehler im MDX-Code

In diesem Zusammenhang ist es relevant zu wissen, dass Storybook zum aktuellen Zeitpunkt leider sehr schlecht auf Syntaxfehler im MDX-Code reagiert. Vergessen Sie beispielsweise die äußeren geschweiften Klammern, stürzt Storybook ab und gibt die folgenden, sehr unkonkreten, Fehlermeldung aus:

```
SyntaxError: Unexpected token, expected "}" (5:14)

    at Object._raise (\user\christoph\sources\buch-12\storybook\
node_modules\@babel\parser\lib\index.js:541:17)
```

Dies kann – insbesondere im Zusammenspiel mit der ungewohnten Syntax – anfangs sehr frustrierend sein und zu mühseligen Fehlersuchen führen. Ich persönlich präferiere deshalb aktuell immer noch die CSF-Syntax für das Schreiben von vollständigen Stories und nutze MDX lediglich für zusätzliche Prosa-Dokumentation.

Ein Blick in die *Storybook*-Oberfläche zeigt, dass Ihre Story nun im Docs-Tab des Reiters Components • Colored Circle (MDX) dargestellt wird.

Abbildung 26 Darstellung der in MDX-geschriebenen Story im Storybook UI

Verwendung von decorators und moduleMetadata in MDX

Neben den in Listing 18 vorgestellten Eigenschaften, erlaubt *MDX* Ihnen selbstverständlich auch die manuelle Konfiguration des `NgModule` einer Story über die Funktion `moduleMetadata` in Verbindung mit der `decorators`-Eigenschaft. In den mitgelieferten Quelltexten finden Sie ein entsprechendes Beispiel in der Datei *panel.stories.mdx*

Input-Bindings in MDX verändern

Möchten Sie in Ihren *MDX*-basierten Stories die Input-Bindings einer Story verändern, so erfolgt dies über die `args`-Eigenschaft der `Story`-Komponente. Listing 19 zeigt die Implementierung der `WithImage`-Story in *MDX*. Beachten Sie auch hier erneut die Verwendung der doppelten geschweiften Klammern.

```
## Mit Hintergrund
Das Setzen eines Hintergrund-Bildes erfolgt über das Input-Binding `imageUrl`
<Story name="WithImage" args={{
    imageUrl: 'http://lorempixel.com/200/200/cats',
}}>
    {Template.bind({})}
</Story>
```

Listing 19 »colored-circle.stories«: Weitere Story mit angepassten Input-Parametern

Wie erwartet wird das Input-Binding `imageUrl` nun mit dem übergebenen Wert beschrieben (siehe Abbildung 27).

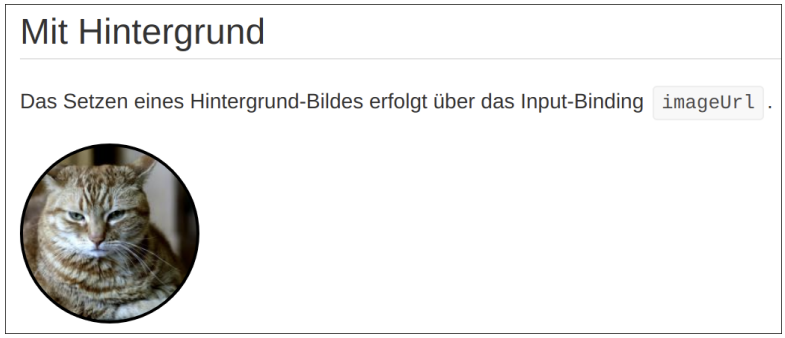


Abbildung 27 Darstellung der zusätzlichen Story mit angepassten Input-Bindings

Story-Templates in MDX überschreiben

Wie im Abschnitt »Explizite Definition von Templates ng-content & Direktiven rendern« beschrieben, erlaubt Storybook Ihnen das für eine Story gerenderte Template manuell festzulegen. In *CSF* erfolgte dies über die `template`-Eigenschaft des `Story`-Objekts:

```
export const Simple: Story<PanelComponent> =  (args: PanelComponent) => ({
  template: `<ch-panel title="Abschnitt 2"> Lorem Ipsum...  </ch-panel>`,
  props: {...args},
});
```

In *MDX* erfolgt diese Definition nun durch die Übergabe eines Konfigurations-Objekts innerhalb des Bodys der `Story`-Komponente (siehe Listing 20).

```
<Story name="CustomHeader"> {{
  template: `
   <ch-panel>
     <ch-panel-header><i>Ich bin ein kursiver Header</i></ch-panel-header>
     Lorem ipsum...
  </ch-panel>`,
}} </Story>
```

Listing 20 »panel.stories.mdx«: Definition von eigenen Story-Templates innerhalb der MDX-Dokumentation

So bietet Ihnen *MDX* auf diese Weise, das gleiche Level an Flexibilität wie das *Component Story Format*.

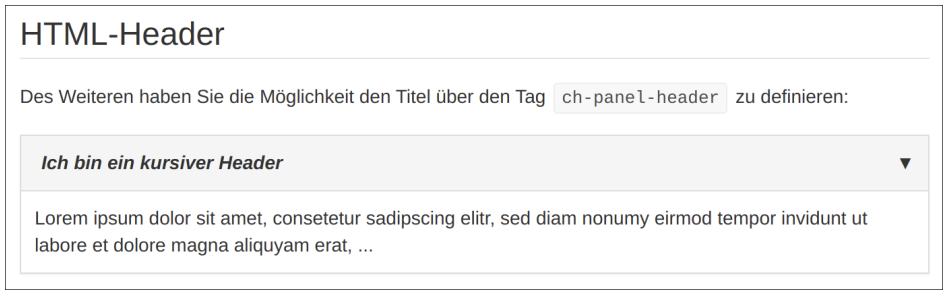


Abbildung 28 Darstellung der Panel-Komponente innerhalb der MDX-Dokumentation

Zusammenfassung und Ausblick

Insbesondere bei der Erstellung von Komponenten-Bibliotheken sowie bei der Implementierung von Anwendungen mit vielen Fremdabhängigkeiten (etwa zu einem Backend) kann *Storybook* Ihnen die Entwicklung von Komponenten stark erleichtern.

Die folgende Liste fasst die wichtigsten Erkenntnisse dieses Kapitels noch einmal kurz zusammen:

- *Storybook* wurde ursprünglich für das Framework *React* entwickelt, unterstützt neben Angular aber mittlerweile eine Vielzahl weiterer Frameworks
- Die Arbeit mit *Storybook* fühlt sich für Angular-Entwickler anfangs oft ungewohnt an, da viele der Begriffe und Techniken aus der React-Welt stammen
- Das *Component Story Format (CSF)* definiert ein Format zu Beschreibung von Stories und Komponenten auf Basis von ES2015 Modulen.
- Möchten Sie unterschiedliche Varianten Ihrer Komponente vorstellen und dafür Story-Bestandteile wiederverwenden bietet sich die Arbeit mit *Template-Stories* an.

- *Controls* erlauben es Input-Bindings interaktiv über die Oberfläche zu verändern.
- Wenn möglich, erkennt *Storybook* automatisch den Typ eines Input-Bindings und bietet in der Oberfläche das korrekte Eingabe-Element (z. B. einen Color-Picker) an.
- Ist dies nicht möglich, können Sie das verwendete Eingabe-Element über die `argTypes` Eigenschaft manuell bestimmen.
- Mit Hilfe von *Actions* können die von Output-Bindings emitteten Werte in der Storybook-Oberfläche dargestellt werden.
- Möchten Sie Direktiven oder `ng-content`-Komponenten in *Storybook* verwenden, müssen Sie das gerenderte Template manuell über die `template`-Eigenschaft der Story beschreiben.
- Die `moduleMetadata`-Funktion ermöglicht es Ihnen, das `NgModule` in dem die Story ausgeführt werden soll, frei zu konfigurieren.
- Möchten Sie Ihre Stories in unterschiedlichen Auflösungen darstellen, bietet *Storybook* Ihnen die Möglichkeit den verwendeten *Viewport* über die Oberfläche auszuwählen.
- Neben vordefinierten *Viewports* ist es ebenfalls möglich eigene Viewports über die zentrale Konfigurationsdatei *preview.js* zu definieren.
- Das *Docs-Plugin* generiert automatisch eine Dokumentations-Seite für jede dokumentierte Komponente.
- Möchten Sie diese Dokumentation mit eigenen Beschreibungen anreichern, so können Sie dies über die `docs`-Eigenschaft der Story erreichen.
- Möchten Sie umfangreichere Dokumentationen verfassen, bietet sich die Verwendung des *MDX-Formats* an.
- *MDX* erlaubt es Ihnen *JSX-Komponenten* in *Markdown* zu integrieren.
- Storybook stellt für die Definition von Stories eigene *JSX-Komponenten* bereit, sodass Sie MDX ebenfalls als Alternative zum *Component Story Format* verwenden können.