
newforms Documentation

Release 0.13.0

Jonny Buchanan

October 23, 2015

1	Getting newforms	3
2	Documentation	5
3	Documentation Contents	7
3.1	Guide Documentation	7
3.2	API Reference	77

An isomorphic JavaScript form-handling library for [React](#).

(Formerly a direct port of the [Django](#) framework's `django.forms` library)

Getting newforms

Node.js Newforms can be used on the server, or bundled for the client using an npm-compatible packaging system such as [Browserify](#) or [webpack](#).

```
npm install newforms
```

```
var forms = require('newforms')
```

Note: By default, newforms will be in development mode. To use it in production mode, set the environment variable `NODE_ENV` to `'production'` when bundling. To completely remove all development mode code, use a minifier that performs dead-code elimination, such as [UglifyJS](#).

Browser bundles The browser bundles expose newforms as a global `forms` variable and expects to find a global `React` variable to work with.

The uncompressed bundle is in development mode, so will log warnings about potential mistakes.

You can find it in the [dist/](#) directory.

Source Newforms source code and issue tracking is on GitHub:

- <https://github.com/insin/newforms>

Documentation

Note: Unless specified otherwise, documented API items live under the `forms` namespace object in the browser, or the result of `require('newforms')` in Node.js.

- **Quickstart** A quick introduction to defining and using newforms Form objects
- **Guide Documentation** An overview of newforms concepts, and guide docs with examples
- **API Reference** Reference guide for the API exposed by newforms

Documentation Contents

3.1 Guide Documentation

3.1.1 Quickstart

A quick introduction to defining and using newforms Form objects.

Design your Form

The starting point for defining your own forms is `Form.extend()`.

Here's a simple (but incomplete!) definition of a type of Form you've probably seen dozens of times:

```
var SignupForm = forms.Form.extend({
  username: forms.CharField(),
  email: forms.EmailField(),
  password: forms.CharField({widget: forms.PasswordInput}),
  confirmPassword: forms.CharField({widget: forms.PasswordInput}),
  acceptTerms: forms.BooleanField({required: true})
})
```

A piece of user input data is represented by a **Field**, groups of related Fields are held in a **Form** and a form input which will be displayed to the user is represented by a **Widget**. Every Field has a default Widget, which can be overridden.

Rendering a Form

Forms provide helpers for rendering labels, user inputs and validation errors for their fields. To get you started quickly, newforms provides a React component which use these helpers to render a basic form structure.

At the very least, you must wrap rendered form contents in a `<form>`, provide form controls such as a submit button and hook up handling of form submission:

```
var Signup = React.createClass({
  render: function() {
    return <form onSubmit={this._onSubmit}>
      <forms.RenderForm form={SignupForm} ref="signupForm"/>
      <button>Sign Up</button>
    </form>
  },
  // ...
});
```

Rendering helpers attach event handlers to the inputs they render, so getting user input data is handled for you.

The `RenderForm` component handles creating a form instance for you, and setting up automatic validation of user input as it's given.

To access this form instance later, make sure the component has a `ref` name.

Handling form submission

The final step in using a Form is validating when the user attempts to submit.

First, use the `ref` name you defined earlier to get the form instance via the `RenderForm` component's `getForm()` method.

Then call the form's `validate()` method to ensure every field in the form is validated against its current user input.

If a Form is valid, it will have a `cleanedData` object containing validated data, coerced to the appropriate JavaScript data type when appropriate:

```
propTypes: {
  onSignup: React.PropTypes.func.isRequired
},

_onSubmit: function(e) {
  e.preventDefault()

  var form = this.refs.signupForm.getForm()
  var isValid = form.validate()
  if (isValid) {
    this.props.onSignup(form.cleanedData)
  }
}
```

Implementing custom validation

There's an obvious [validation](#) not being handled by our form: what if the passwords don't match?

This is a cross-field validation. To implement custom, cross-field validation add a `clean()` method to the Form definition:

```
clean: function() {
  if (this.cleanedData.password &&
      this.cleanedData.confirmPassword &&
      this.cleanedData.password !== this.cleanedData.confirmPassword) {
    throw forms.ValidationError('Passwords do not match.')
  }
}
```

Live demo

This is the React component we defined above being used by another component which passes an `onSignup()` callback to receive and display submitted data:

3.1.2 Overview

Newforms takes care of a number of common form-related tasks. Using it, you can:

- Display a form with automatically generated form widgets.
- Automatically keep a JavaScript object in sync with current user input.
- Check user input against a set of validation rules.
- Update form display with validation errors.
- Convert valid user input to the relevant JavaScript data types.

Concepts

The core concepts newforms deals with are:

Widgets Widgets create `ReactElement` objects for form inputs.

For example, a `Select` widget knows which `<option>` values and labels it should generate and how to generate a `<select>` with the `<option>` corresponding to given user input data marked as selected.

Fields A Field holds metadata about displaying a form input and validating a piece of user input. Its metadata is the source for:

- Configuring a suitable Widget to generate a form input, with a label and help text.
- Validating user input and providing an appropriate error message when it's invalid.
- Converting valid user input to an appropriate JavaScript data type.

For example, an `IntegerField` makes sure that its user input data is a valid integer, is valid according to any additional rules defined – such as `minValue` – and converts valid user input to a JavaScript `Number`. By default, it configures a `NumberInput` widget to display an `<input type="number">` for input.

Forms Forms group related Fields together, using them to validate user input and providing helpers for displaying them as HTML.

Forms drive the validation process, holding raw user input data, validation error messages and “cleaned” data which has been validated and type-converted.

Form objects

Form constructors are created by extending `forms.Form` and declaratively specifying field names and metadata:

```
var ContactForm = forms.Form.extend({
  subject: forms.CharField({maxLength: 100}),
  message: forms.CharField(),
  sender: forms.EmailField(),
  ccMyself: forms.BooleanField({required: false})
})
```

A form is composed of Field objects. In this case, our form has four fields: `subject`, `message`, `sender` and `ccMyself`. `CharField`, `EmailField` and `BooleanField` are just three of the available field types – a full list can be found in [Form fields](#).

- A form with no user input data will render as empty or will contain any default values it was configured with.
- Once a form has user input data, it can validate it. If a form is rendered with invalid input data, it can include error messages telling the user what to correct.

Processing input data with a Form

When a form is given valid input data, the successfully validated form data will be in the `form.cleanedData` object. This data will have been converted into JavaScript types for you, where necessary.

In the above example, `ccMyself` will be a boolean value. Likewise, fields such as `IntegerField` and `DateField` convert values to a JavaScript Number and Date, respectively.

Displaying a Form

Newforms provides a React component which implements default rendering of Form objects.

This only provides rendering of the form’s own fields; it’s up to you to provide the surrounding `<form>` element, submit buttons etc:

```
render: function() {
  return <form action="/contact" method="POST" onSubmit={this.onSubmit}>
    <forms.RenderForm form={ContactForm}/>
    <div>
      <input type="submit" value="Submit"/>{' '}
      <input type="button" value="Cancel" onClick={this.onCancel}/>
    </div>
  </form>
}
```

RenderForm will output each form field and accompanying label wrapped in a `<div>`. Here’s the output for our example component:

```
<form action="/contact" method="POST">
  <div>
    <div><label for="id_subject">Subject:</label> <input maxlength="100" type="text" name="subject" id="id_subject"></div>
    <div><label for="id_message">Message:</label> <input type="text" name="message" id="id_message"></div>
    <div><label for="id_sender">Sender:</label> <input type="email" name="sender" id="id_sender"></div>
    <div><label for="id_ccMyself">Cc myself:</label> <input type="checkbox" name="ccMyself" id="id_ccMyself"></div>
  </div>
  <div><input type="submit" value="Submit"> <input type="button" value="Cancel"></div>
</form>
```

Note that each form field has an `id` attribute set to `id_<field-name>`, which is referenced by the accompanying label tag. You can *customise the way in which labels and ids are generated*.

You can customise the output of `RenderForm` by passing certain props or a child React component. See the *documentation for custom rendering* for more info.

You can also *completely customise form display* using rendering helpers for each field.

“Isomorphic”

Newforms was developed to be independent of the DOM, so it can be used on the server as well as the browser. This allows you to use it when pre-rendering initial HTML on the server, to be rehydrated and reused by React on the client side.

User inputs generated by newforms have unique `name` attributes and its handling of input data is compatible with POST data which would be submitted for the user inputs it generates, so you can also use it to validate regular form submissions on the server and redisplay with errors, should you need to.

This makes newforms suitable for creating apps which use good old HTTP round tripping, or for JavaScript apps which need to be capable of falling back to being usable as regular forms ‘n links webapps in any device.

For an example of using newforms this way, check out the [Isomorphic Lab demo](#) and its [source on GitHub](#).

3.1.3 React Components

New in version 0.10.

To help you get started quickly, newforms provides some React components to handle instantiation and rendering of Forms and FormSets.

For the basic scenario of displaying form fields in the order they were defined, these may be all you need to handle rendering your forms.

RenderForm

This component renders a [Form](#) as a list of “rows” – one for each field.

It handles the generic use case for form rendering:

- Whole-form error messages are displayed at the top.
- Each visible field in the form is displayed in the order fields were defined.

It can also take care of some of the details of creating a Form instance and re-rendering when the form’s state changes for you.

```
<RenderForm form={MyForm} ref="myForm"/>
```

Form options

If *form construction options* are passed as props to RenderForm, they will be passed on to the Form constructor when creating an instance.

For example, if you need to display more than one of the same Form, you need to specify a *prefix* to give them a unique namespace, so pass a prefix prop like so:

```
var ParentForm = forms.Form.extend({
  name: forms.CharField(),
  dob: forms.DateField({label: 'Date of birth'})
})
```

```
<fieldset>
  <legend>Parent 1</legend>
  <RenderForm form={ParentForm} prefix="parent1"/>
</fieldset>
<fieldset>
  <legend>Parent 2</legend>
  <RenderForm form={ParentForm} prefix="parent2"/>
</fieldset>
```

Getting the Form instance

When passing RenderForm a Form constructor, give it a *ref* prop so you can use its `getForm()` method to access the Form instance it manages for you.

For example, when handling submission of a form:

```
render: function() {
  return <form onSubmit={this._onSubmit}>
    <forms.RenderForm form={MyForm} ref="myForm"/>
    <button>Submit</button>
  </form>
},

_onSubmit: function(e) {
  e.preventDefault()
  var form = this.refs.form.getForm()
  var isValid = form.validate()
  if (isValid) {
    // ..
  }
}
```

Other rendering scenarios

For the sake of being a complete default rendering implementation, `RenderForm` also handles some less common scenarios:

- Displaying error messages related to hidden fields.
- Rendering hidden fields.
- Displaying a progress indicator if the form has asynchronous whole-form validation pending completion.

RenderForm props

Note: A `ReactCompositeComponent` is what you get back when you call `React.createClass()`

form

type `Form` or `Function` (a `Form` constructor)

The `Form` to be rendered – can be a constructor or an instance.

If you pass a `Form` constructor, the component will instantiate it for you. *Form construction options* may also be passed to `RenderForm` as additional props.

If you pass a `Form` instance, make sure you set up its *`onChange()`* in such a way that it will also re-render the `<RenderForm/>` component when the form changes.

component

type `ReactCompositeComponent` or `String` (an HTML tag name)

The component used to wrap all the form's rows – defaults to `'div'`.

className

type `String`

If provided, this prop will be passed to the wrapper component containing all the form's rows.

row

type `ReactCompositeComponent`

The component used to render each form row – defaults to *`FormRow`*.

rowComponent

type `ReactCompositeComponent` or `String` (an HTML tag name)

The tag name or component used to wrap each form row. Defaults to `'div'`.

This is passed as a `component` prop to the component in the `row` prop.

progress

type `ReactCompositeComponent` or `Function`

Used to render what's displayed if the form has an `async clean()` method which is pending completion.

This will also be passed to the component in the `row` prop when rendering.

Form construction options All the *options which be passed when instantiating a Form* can be passed as props to `RenderForm` for use when you pass a Form constructor as the `form` prop.

RenderForm methods

getForm() Returns the Form instance being rendered by the component.

FormRow

This component handles rendering a single form “row”. *RenderForm* uses this to render rows by default; it will either:

1. Wrap some given content (such as a list of error messages) as a row, or:
2. Use a field's *rendering helper* to generate a row for the field, with a label, user input, error messages and help text, as necessary.

FormRow props**content**

type `Any`

If given, will be used for the entire contents of the row.

bf

type `BoundField()`

If given and `content` is not given, this Field rendering helper will be used to create contents for a Field row.

component

type `ReactCompositeComponent` or `String` (an HTML tag name)

The component used to wrap the row contents. Defaults to `'div'`.

className

type `String`

If provided, this prop will be passed to the wrapper component for the row.

hidden

type `Boolean`

If `true`, the row container `ReactElement` will be given a `display: none` style – defaults to `false`.

progress

type `ReactCompositeComponent` or `Function`

Used to render what's displayed if the form has `async clean<Field>()` method which is pending completion.

RenderFormSet

This component handles the generic use case for `FormSet` rendering, using `RenderForm` to render each form in a formset one after the other.

It can also take care of some of the details of creating a `FormSet` and re-rendering when form state changes.

```
<RenderFormSet form={MyForm} extra="3" ref="myFormset"/>
<RenderFormSet formset={MyFormSet} ref="myFormset"/>
```

RenderFormSet props**form**

type `Function` (a `Form` constructor)

If you pass a `Form` constructor, the component will instantiate a `FormSet` for you.

`FormSet` constructor options may be passed as additional props to `RenderFormSet`.

Note: When a `form` prop is passed, use of the `formset` prop changes. If also provided, it must be a `FormSet` constructor to be extended from.

formset

type `FormSet` or `Function` (a `FormSet` constructor)

The `FormSet` to be rendered – can be a constructor or an instance.

If you pass a `FormSet` constructor, the component will instantiate it for you. *FormSet construction options* may also be passed to `RenderFormSet` as additional props.

If you pass a `FormSet` instance, make sure you set up its *onChange()* in such a way that it will also re-render the `<RenderFormSet/>` component when one of its forms changes.

component

type `ReactCompositeComponent` or `String` (an HTML tag name)

The component used to wrap the formset's contents. Defaults to `'div'`.

className

type `String`

If provided, this prop will be passed to the wrapper component for the formset.

formComponent

type `ReactCompositeComponent` or `String` (an HTML tag name)

The component used to wrap each form. Defaults to `'div'`.

This is passed as a `component` prop to *RenderForm*.

row & rowComponent These are *as defined above* for `RenderForm`, which they are passed to.

progress

type `ReactCompositeComponent` or `Function`

Used to render what's displayed if the formset has an `async clean()` method which is pending completion.

This will also be passed to *RenderForm*.

useManagementForm

type `Boolean`

If `true`, hidden fields from the `FormSet`'s management form will be rendered. Defaults to `false`.

These fields are usually only required if you will be performing a regular form submission which will be processed by newforms on the server.

RenderFormSet methods

getFormset() Returns the `FormSet` instance being rendered by the component.

Custom rendering with props

The bundled React components offer a degree of customisation via their props.

You can use the `component`, `className` and `rowComponent` props to customise the containers:

```
<RenderForm form={ParentForm}
  component="ul"
  className="parent"
  rowComponent="li"
  autoId={false}
/>
```

Which renders as:

```
<ul class="parent">
  <li>Name: <input type="text" name="name"></li>
  <li>Date of birth: <input type="text" name="dob"></li>
</ul>
```

You can also customise how form rows are generated by passing a custom React component to the `row` prop.

```
<RenderForm form={ParentForm} row={MySpecialFormRow}/>
```

Note: Keep in mind when implementing a custom row component that it will receive props as per those described for *FormRow*.

Custom rendering with a child component

If you want to implement custom form rendering with your own React component while still making use of `RenderForm` to instantiate the form and set up automatic validation and redisplay, pass a component as the only child of `RenderForm`.

Warning: Passing more than one child component to `RenderForm` will result in an `Error`.

`RenderForm` will then clone your component and pass the Form instance it manages as a `form` prop.

For example, this is how `newforms-gridforms` implements a custom grid layout:

```
<RenderForm form={ParentForm}>
  <GridForm>
    <Section name="Parent">
      <Row>
        <Field name="name"/>
        <Field name="dob"/>
      </Row>
    </Section>
  </GridForm>
</RenderForm>
```

Changed in version 0.13: The Form instance is also available via React's `context feature` as a `form` context variable, allowing you to access the Form instance in nested custom rendering components without having to manually pass it down via props.

Custom async progress rendering

By default, when *async validation* is in progress, each of the React components `newforms` provides will render a `<progress>` element with fallback “Validating...” text. However, the `<progress>` element doesn't currently lend itself to extensive customisation via CSS, especially cross-browser.

To customise this, each component takes a `progress` prop which can take a function or React component which will be used to indicate an in-progress async validation.

For example, either of the following could be passed as the `progress` prop to display a spinner image instead:

```
var InProgress = React.createClass({
  render() {
    return <span>
       Validating&hellip;
    </span>
  }
})

function inProgress() {
  return <span>
     Validating&hellip;
  </span>
}
```

```
<RenderForm form={MyForm} ref="myForm" progress={InProgress}/>
```

3.1.4 Interactive Forms with React

This section focuses on API and patterns of usage that are applicable to using `newforms` to create interactive forms in a React component in the browser.

Other documentation sections use the API for providing user input when creating a Form instance, to demonstrate behaviour and features based on user input in a concise way.

However, this API is more typical of using newforms on the server or as a standalone validator. When working with Forms in the client, user input is more often taken one field at a time using the `onChange` event.

Provide a containing `<form>`

Warning: You **must** provide a `<form>` to contain fields you're rendering with newforms.

At the time of documenting (version 0.10), you **must** provide a `<form>` to contain fields you're rendering with newforms. It's likely that you'll want one anyway to make use of its `onSubmit` event for *Final Form validation*.

React's virtual DOM doesn't provide a means of obtaining values from form inputs. To do this for you, newforms must reach into the real DOM when an `onChange` event fires on one of the inputs it rendered for you.

Forms currently make use of the real DOM's `form.elements` collection to simplify retrieving values for fields which render as multiple inputs, such as a `MultipleChoiceField` which uses a `CheckboxSelectMultiple` widget.

`noValidate`

Some newforms Widgets render HTML5 input types which, by default, perform their own validation.

When you're using newforms with React for interactive validation, this can give your forms an inconsistent feeling as the native inputs have their own way of presenting validation errors. Consider adding a `noValidate` prop to your `<form>` to disable native validation if this starts to feel like an issue:

```
<form onSubmit={this.handleSubmit} noValidate>
```

If you're creating an isomorphic application, you might want to conditionally use `noValidate` so there's some validation while the page is still initially loading, or for scenarios when JavaScript isn't available on the client:

```
getInitialState() {
  return {client: false}
},

componentDidMount() {
  this.setState({client: true})
},

render() {
  return <form onSubmit={this.handleSubmit} noValidate={!this.state.client}>
    ...
  }
}
```

Creating Forms and FormSets

There are a number of client-specific options available when creating an instance of a Form to use in a React component. The same options apply when creating FormSets, as they use them to handle creation of Form instances for you.

Form state and `onChange()`

While a Form is not itself a React component, it is stateful. Its `data`, `errors()` and `cleanedData` properties will be changed when the user makes changes and their input is stored and validated.

In order to update display of its containing React component, a Form will call a given `onChange()` callback each time user input is taken or validation is performed in response to a user changing input data.

Note: The details of setting up `onChange` are handled for you when rendering a Form by passing its constructor to a `<RenderForm/>` component.

Typically, this function will just force its React component to update, for example:

```
getInitialState: function() {
  return {
    form: new ContactForm({onChange: this.onFormChange})
  },
},

onFormChange: function() {
  this.forceUpdate()
}
```

Passing an `onChange` callback will also automatically configure interactive validation of each form input as it's updated by the user. See below for details of what that entails and how to configure it.

Warning: Due to the way controlled components work in React, if you are using *Controlled user inputs* and you do not pass an `onChange()` callback, your form inputs will be read-only! The development version of newforms will warn you if this happens.

Interactive Form validation

To validate individual input fields as the user interacts with them, you can pass a `validation` argument when instantiating a Form or Field; passing a `validation` argument when instantiating a Form sets up interactive validation for every Field in it.

Form 'auto' validation

Important: When you pass an `onChange` callback to a Form, its `validation` mode is automatically implied to be `'auto'`:

```
var form = new SignupForm({onChange: this.onFormChange})
```

When the validation mode is `'auto'`:

- Text fields are validated using the `onChange` and `onBlur` events, with a debounce delay of 369ms applied to `onChange` between the last change being made and validation being performed.
- Other input fields are validated as soon as the user interacts with them.

Note: React normalises the `onChange` event in text inputs to fire after every character which is entered.

'auto' example form

Let's use a standard signup form as an example:

```

var SignupForm = forms.Form.extend({
    email: forms.EmailField(),
    password: forms.CharField({widget: forms.PasswordInput}),
    confirm: forms.CharField({label: 'Confirm password', widget: forms.PasswordInput}),
    terms: forms.BooleanField({
        label: 'I have read and agree to the Terms and Conditions',
        errorMessages: {required: 'You must accept the terms to continue'}
    }),

    clean: function() {
        if (this.cleanedData.password && this.cleanedData.confirm &&
            this.cleanedData.password != this.cleanedData.confirm) {
            throw forms.ValidationError('Passwords do not match.')
        }
    }
})

```

Note that this Form defines a *clean()* function for cross-field validation. In addition to validating the field which just changed, user input will also trigger cross-field validation by calling `clean()`. This function must always be written defensively regardless of whether full or partial validation is being run, as it can't assume that any of the `cleanedData` it validates against will be present due to the possibility of missing or invalid user input.

Field validation

Fields also accept a `validation` argument – validation defined at the field level overrides any configured at the Form level, so if you want to use interaction validation only for certain fields, or to opt fields out when validation has been configured at the Form level, use the `validation` argument when defining those fields.

validation options

`'manual'`

This is the default option, which disables interactive validation.

You're only likely to need to use this if you're opting specific fields out of form-wide interactive validation.

validation object

Interactive validation can be specified as an object with the following properties:

on The name of the default event to use to trigger validation on text input fields. This can be specified with or without an `'on'` prefix. If validation should be triggered by multiple events, their names can be passed as a space-delimited string or a list of strings.

For example, given `validation: {on: 'blur'}`, text input validation will be performed when the input loses focus after editing.

onChangeDelay A delay, in milliseconds, to be used to debounce performing of validation when using the `onChange` event, to give the user time to enter input without distracting them with error messages or other display changes around the input while they're still typing.

`'auto'`

The behaviour of `'auto'` validation is *documented above*. It's equivalent to passing:

```
validation: {on: 'blur change', onChangeDelay: 369}
```

Any event name

If you pass any other string as the `validation` argument, it will be assumed to be an event name, so the following lines are equivalent:

```
validation: 'blur'
validation: {on: 'blur'}
```

Final Form validation

Whether or not you've given your Form an `onChange` callback, Forms will still automatically update their data object with user input as the user interacts with each input field. Even if all fields have been used and are valid, the user still has to signal their intent to submit before any final validation can be performed.

Validating final form submission is left in your hands, as newforms doesn't know (or care, sorry!) what you ultimately want to do with the `cleanedData` it creates for you.

This is typically implemented by hooking into a `<form>`'s `onSubmit` event and calling `form.validate()` to validate the entire user input.

```
onSubmit: function(e) {
  e.preventDefault()
  var form = this.state.form
  var isValid = form.validate()
  if (isValid) {
    this.props.processContactData(form.cleanedData)
  }
}
```

Tip: Forms represent groups of related Fields and don't necessarily have to model the content of the entire `<form>`. Use as many as you like, but don't forget to use *prefixes* when necessary to avoid input field name and id clashes.

Controlled user inputs

By default, newforms generates *uncontrolled React components* for user inputs, which can provide initial values but require manual updating via the DOM should you wish to change the displayed values from code.

If you need to programatically update the values displayed in user inputs after their initial display, you will need to use *controlled React components*.

You can do this by passing a `controlled` argument when constructing the Form or individual Fields you wish to have control over:

```
var form = new SignupForm({controlled: true, onChange: this.onFormChange})
```


Controlled components created by newforms reflect the values held in `form.data`. It's recommended that you call `form.setData()` or `form.updateData()` to update `form.data`, as they handle transitioning from initial display of data to displaying user input and will also call `onChange()` for you, to trigger re-rendering of the containing React component.

controlled example Form

An example of reusing the same controlled Form to edit a bunch of different objects which have the same fields.

First, define a form:

```
var PersonForm = forms.Form.extend({
  name: forms.CharField({maxLength: 100}),
  age: forms.IntegerField({minValue: 0, maxValue: 115}),
  bio: forms.CharField({widget: forms.Textarea})
})
```

When creating the form in our example React component, we're passing `controlled: true`:

```
getInitialState: function() {
  return {
    form: new PersonForm({
      controlled: true,
      onChange: this.forceUpdate.bind(this)
    }),
    editing: null,
    people: [/* ... */]
  }
}
```

To update what's displayed in the form, we have a `handleEdit` function in our React component which is calling `form.reset()` to put the form back into its initial state, with new initial data:

```
handleEdit: function(personIndex) {
  this.state.form.reset(this.state.people[personIndex])
  this.setState({editing: personIndex})
}
```

Rendering Forms

One of the benefits of using React is that display logic really is Just JavaScript. Reusable pieces can be extracted into functions, or React components, or a configurable object of some sort or... whatever your programming heart desires.

Newforms gives you a rendering helper – called a `BoundField` – for each field, which has access to the `Field`, its `Widget` and its `Form`, which collectively have access to all the metadata and user input data it needs to render the field. It uses these to implement rendering helper methods, which are available for you to use in your react components.

`BoundFields`, their most useful properties and examples of their use are covered in [Customising Form display](#) and the complete [BoundField API](#) is documented.

3.1.5 Customising Form display

The [React Components](#) provided by newforms can help you get started quickly and offer a degree of customisation, but you can completely customise the way a form is presented by rendering it yourself.

To assist with rendering, we introduce another concept which ties together `Widgets`, `Fields` and `Forms`:

BoundField

BoundField A `BoundField()` is a helper for rendering HTML content for – and related to – a single Field.

It ties together the Field itself, the field's configured Widget, the name the field is given by the Form, and the raw user input data and validation errors held by a Form.

BoundFields provide properties and functions for using these together to render the different components required to display a field – its label, form inputs and validation error messages – as well as exposing the constituent parts of each of these should you wish to fully customise every aspect of form display.

Forms provide a number of methods for creating BoundFields. These are:

- `form.boundFieldsObj()` – returns an object whose properties are the form's field names, with BoundFields as values.
- `form.boundFields()` – returns a list of BoundFields in their form-defined order.
- `form.boundField(fieldName)` – returns the BoundField for the named field.

Every object which can generate `ReactElement` objects in newforms has a default `render()` method – for BoundFields, the default `render()` for a non-hidden field calls `asWidget()`, which renders the Widget the field is configured with.

A selection of the properties and methods of a BoundField which are useful for custom field rendering are listed below. For complete details, see the [BoundField API](#).

Useful BoundField properties

bf.name The name of the field in the form.

bf.htmlName The name the field will be represented by when rendered. If each Form and FormSet being used to render user inputs has a unique *prefix*, this is guaranteed to be a unique name.

As such, it's a good candidate if you need a unique *key* prop for a React component related to each field.

bf.label The label text for the field, e.g. 'Email address'.

bf.helpText Any help text that has been associated with the field.

bf.field The `Field()` instance from the form, that this `BoundField()` wraps. You can use it to access field properties directly.

Newforms also adds a *custom property* to the Field API – you can pass this argument when creating a field to store any additional, custom metadata you want to associate with the field for later use.

Useful BoundField methods

bf.errors() Gets an object which holds any validation error messages for the field and has a default rendering to a `<ul class="errorlist">`.

bf.errorMessage() Gets the first validation error message for the field as a String, or `undefined` if there are none, making it convenient for conditional display of error messages.

bf.idForLabel() Generates the id that will be used for this field. You may want to use this in lieu of `labelTag()` if you are constructing the label manually.

bf.labelTag() Generates a `<label>` containing the field's label text, with the appropriate `htmlFor` property.

bf.helpTextTag() By default, generates a `` containing the field's help text if it has some configured, but this can be configured with arguments.

New in version 0.10.

bf.status() Gets the current validation status of the field as a string, one of:

- 'pending' – has a pending async validation.
- 'error' – has validation errors.
- 'valid' – has neither of the above and data present in `form.cleanedData`.
- 'default' – none of the above (likely hasn't been interacted with or validated yet).

New in version 0.10.

bf.value() Gets the value which will be displayed in the field's user input.

boundFields() example

Using these, let's customise rendering of our `ContactForm`. Rendering things in React is just a case of creating `ReactElement` objects, so the full power of JavaScript and custom React components are available to you.

For example, let's customise rendering to add a CSS class to our form field rows and to put the checkbox for the `ccMyself` field inside its `<label>`:

```
function renderField(bf) {
    var className = 'form-field'
    if (bf.field instanceof forms.BooleanField) {
        return <div className={className}>
            <label>{bf.render()} {bf.label}</label>
            {bf.helpTextTag()} {bf.errors().render()}
        </div>
    }
    else {
        return <div className={className}>
            {bf.labelTag()} {bf.render()}
            {bf.helpTextTag()} {bf.errors().render()}
        </div>
    }
}
```

We still don't need to do much work in our component's `render()` method:

```
render: function() {
    return <form action="/contact" method="POST">
        {this.state.form.boundFields().map(renderField)}
        <div>
            <input type="submit" value="Submit"/>{' '}
            <input type="button" value="Cancel" onClick={this.onCancel}/>
        </div>
    </form>
}
```

Its initial rendered output is now:

```
<form action="/contact" method="POST">
  <div class="form-field"><label for="id_subject">Subject:</label> <input maxlength="100" type="text"
  <div class="form-field"><label for="id_message">Message:</label> <input type="text" name="message"
  <div class="form-field"><label for="id_sender">Sender:</label> <input type="email" name="sender" id="id_sender">
```

```
<div class="form-field"><label for="id_ccMyself"><input type="checkbox" name="ccMyself" id="id_ccMyself">
<div><input type="submit" value="Submit"> <input type="button" value="Cancel"></div>
</form>
```

boundFieldsObj() example

The following Form and FormSet will be used to take input for a number of items to be cooked:

```
var ItemForm = forms.Form.extend({
    name: Forms.CharField(),
    time: Forms.IntegerField(),
    tend: Forms.ChoiceField({required: false, choices: ['', 'Flip', 'Rotate']})
})

var ItemFormSet = forms.FormSet.extend({form: ItemForm, extra: 3})
```

The list of item forms will be presented as a `<table>` for alignment and compactness. We could use `boundFields()` as above and loop over each form's fields, creating a `<td>` for each one, but what if we wanted to display a unit label alongside the "time" field and dynamically display some extra content alongside the "tend" field?

If every field needs to be rendered slightly differently, or needs to be placed individually into an existing layout, `boundFieldsObj()` provides a convenient way to access the form's `BoundFields` by field name:

```
<tbody>
  {itemFormset.forms().map(function(itemForm, index) {
    var fields = itemForm.boundFieldsObj()
    return <tr>
      <td>{fields.name.render()}</td>
      <td>{fields.time.render()} mins</td>
      <td>
        {fields.tend.render()}
        {fields.tend.value() && ' halfway'}
      </td>
    </tr>
  })}
</tbody>
```

3.1.6 Forms

Note: In code examples which display HTML string output, we use a `reactHTML()` function as shorthand convention for rendering a `ReactElement` as a static HTML string.

Initial input data

When constructing a `Form()` instance, whether or not you pass input data determines the behaviour of the form's initial render.

- If a user input data object is given, initial rendering will trigger validation when it tries to determine if there are any error messages to be displayed.

This is typically how user input is passed to the form when using newforms on the server to validate a POST request's submitted data.

- If data is not given, validation will not be performed on initial render, so the form can render with blank inputs or display any default initial values that have been configured, without triggering validation.

To create a Form instance , simply instantiate it:

```
var f = new ContactForm()
```

To create an instance with input data, pass `data` as an option argument, like so:

```
var data = {
  subject: 'hello'
, message: 'Hi there'
, sender: 'foo@example.com'
, ccMyself: true
}
var f = new ContactForm({data: data})
```

In this object, the property names are the field names, which correspond to the names in your `Form` definition. The values are the data you're trying to validate. These will usually be strings, but there's no requirement that they be strings; the type of data you pass depends on the `Field()`, as we'll see in a moment.

Data can also be set on a Form instance, which triggers validation, returning the validation result:

```
var isValid = f.setData(data)
```

`form.isInitialRender`

If you need to distinguish between the type of rendering behaviour a form instance will exhibit, check the value of the form's `form.isInitialRender` property:

```
var f = new ContactForm()
print(f.isInitialRender)
// => true
f = new ContactForm({data: {subject: 'hello'}})
print(f.isInitialRender)
// => false
```

A form given an *empty* data object will still be considered to have user input and will trigger validation when rendered:

```
var f = new ContactForm({data: {}})
print(f.isInitialRender)
// => false
```

Using forms to validate data

Server or standalone validation

The primary task of a `Form` object is to validate data. With a bound `Form` instance, call the `Form#isValid()` method to run validation and return a boolean designating whether the data was valid:

```
var data = {
  subject: 'hello'
, message: 'Hi there'
, sender: 'foo@example.com'
, ccMyself: true
}
var f = new ContactForm({data: data})
print(f.isValid())
// => true
```

Let's try with some invalid data. In this case, `subject` is blank (an error, because all fields are required by default) and `sender` is not a valid email address:

```
var data = {
  subject: 'hello',
  message: 'Hi there'
, sender: 'invalid email address'
, ccMyself: true
}
var f = new ContactForm({data: data})
print(f.isValid())
// => false
```

`form.errors()` returns an *ErrorObject()* containing error messages:

```
f.errors().asText()
/* =>
* subject
* This field is required.
* sender
* Enter a valid email address.
*/
```

You can access `form.errors()` without having to call `Form.isValid()` first. The form's data will be validated the first time you either call `form.isValid()` or `form.errors()`.

The validation routines will only get called once for a given set of data, regardless of how many times you call `form.isValid()` or `form.errors()`. This means that if validation has side effects, those side effects will only be triggered once per set of input data.

Client validation

On the client-side, the user's input is held in form DOM inputs, not a tidy JavaScript object as in the above examples (whereas if you're handling a request on the server, the request body serves this purpose).

Regardless of whether or not you're using event-based validation, the form's input data will be updated as the user fills it in. To force the form to fully validate, call `form.validate()`:

```
// Form creation in a React component's getInitialState()
var form = new ContactForm()

// Validation in an onSubmit event handler
var isValid = this.state.form.validate()
```

Dynamic initial values

Use `form.initial` to declare the initial value of form fields at runtime. For example, you might want to fill in a username field with the username of the current session.

To do this, pass an `initial` argument when constructing the form. This argument, if given, should be an object mapping field names to initial values. You only have to include the fields for which you're specifying an initial value, for example:

```
var f = new ContactForm({initial: {subject: 'Hi there!'}})
```

Where both a Field and Form define an initial value for the same field, the Form-level `initial` gets precedence:

```

var CommentForm = forms.Form.extend({
    name: forms.CharField({initial: 'prototype'}),
    url: forms.URLField(),
    comment: forms.CharField()
})

var f = new CommentForm({initial: {name: 'instance'}, autoId: false})
print(reactHTML(<RenderForm form={f}/>))
/* =>
<div>
  <div>Name: <input type="text" name="name" value="instance"></div>
  <div>Url: <input type="url" name="url"></div>
  <div>Comment: <input type="text" name="comment"></div>
</div>
*/

```

Accessing the fields from the form

You can access the fields of a Form instance from its `fields` attribute:

```

print(f.fields)
// => {name: [object CharField], url: [object URLField], comment: [object CharField]}

```

You can alter fields of a Form instance:

```

f.fields.name.label = 'Username'
print(reactHTML(<RenderForm form={f}/>))
/* =>
<div>
  <div>Username: <input type="text" name="name" value="instance"></div>
  <div>Url: <input type="url" name="url"></div>
  <div>Comment: <input type="text" name="comment"></div>
</div>
*/

```

Warning: don't alter `baseFields` or every subsequent form instance will be affected:

```

f.baseFields.name.label = 'Username'
var anotherForm = new CommentForm({autoId: false})
print(reactHTML(<RenderForm form={anotherForm}/>))
/* =>
<div>
  <div>Username: <input type="text" name="name" value="prototype"></div>
  <div>Url: <input type="url" name="url"></div>
  <div>Comment: <input type="text" name="comment"></div>
</div>
*/

```

Accessing “clean” data

Each field in a Form is responsible not only for validating data, but also for “cleaning” it – normalising it to a consistent format. This allows data for a particular field to be input in a variety of ways, always resulting in consistent output.

Once a set of input data has been validated, you can access the clean data via a form's `cleanedData` property:

```

var data = {
    subject: 'hello'
}

```

```
, message: 'Hi there'
, sender: 'foo@example.com'
, ccMyself: true
}
var f = new ContactForm({data: data})
print(f.isValid())
// => true
print(f.cleanedData)
// => {subject: 'hello', message: 'Hi there', sender: 'foo@example.com', ccMyself: true}
```

If input data does *not* validate, `cleanedData` contains only the valid fields:

```
var data = {
    subject: ''
, message: 'Hi there'
, sender: 'foo@example.com'
, ccMyself: true
}
var f = new ContactForm({data: data})
print(f.isValid())
// => false
print(f.cleanedData)
// => {message: 'Hi there', sender: 'foo@example.com', ccMyself: true}
```

`cleanedData` will only contain properties for fields defined in the form, even if you pass extra data:

```
var data = {
    subject: 'Hello'
, message: 'Hi there'
, sender: 'foo@example.com'
, ccMyself: true
, extraField1: 'foo'
, extraField2: 'bar'
, extraField3: 'baz'
}
var f = new ContactForm({data: data})
print(f.isValid())
// => false
print(f.cleanedData) // Doesn't contain extraField1, etc.
// => {subject: 'hello', message: 'Hi there', sender: 'foo@example.com', ccMyself: true}
```

When the Form is valid, `cleanedData` will include properties for all its fields, even if the data didn't include a value for some optional fields. In this example, the data object doesn't include a value for the `nickName` field, but `cleanedData` includes it, with an empty value:

```
var OptionalPersonForm = forms.Form.extend({
    firstName: forms.CharField(),
    lastName: forms.CharField(),
    nickName: forms.CharField({required: false})
})
var data {firstName: 'Alan', lastName: 'Partridge'}
var f = new OptionalPersonForm({data: data})
print(f.isValid())
// => true
print(f.cleanedData)
// => {firstName: 'Alan', lastName: 'Partridge', nickName: ''}
```

In the above example, the `cleanedData` value for `nickName` is set to an empty string, because `nickName` is a `CharField`, and `CharFields` treat empty values as an empty string.

Each field type knows what its “blank” value is – e.g., for `DateField`, it’s `null` instead of the empty string. For full details on each field’s behaviour in this case, see the “Empty value” note for each field in the [Built-in Field types](#) documentation.

You can write code to perform validation for particular form fields (based on their name) or for the form as a whole (considering combinations of various fields). More information about this is in [Form and Field validation](#).

Updating a form’s input data

`form.setData()`

To replace a Form’s entire input data with a new set, use `form.setData()`.

This will also trigger validation – updating `form.errors()` and `form.cleanedData`, and returning the result of `form.isValid()`:

```
var f = new ContactForm()
// ...user inputs data...
var data = {
  subject: 'hello'
, message: 'Hi there'
, sender: 'foo@example.com'
, ccMyself: true
}
var isValid = f.setData(data)
print(f.isInitialRender)
// => false
print(isValid)
// => true
```

`form.updateData()`

To partially update a Form’s input data, use `form.updateData()`.

This will trigger validation of the fields for which new input data has been given, and also any form-wide validation if configured.

It doesn’t return the result of the validation it triggers, since the validity of a subset of fields doesn’t tell you whether or not the entire form is valid.

If you’re performing partial updates of user input (which is the case if individual fields are being validated `onChange`) and need to check if the entire form is valid *without* triggering validation errors on fields the user may not have reached yet, use `Form#isComplete()`:

```
var f = new ContactForm()
f.updateData({subject: 'hello'})
print(f.isComplete())
// => false
f.updateData({message: 'Hi there'})
print(f.isComplete())
// => false
f.updateData({sender: 'foo@example.com'})
print(f.isComplete())
// => true
```

Note that `form.isComplete()` returns `true` once all required fields have valid input data.

Outputting forms as HTML

Forms provide [helpers for rendering their fields](#) but don't know how to render themselves. The `RenderForm` [React component](#) uses these helpers to provide a default implementation of rendering a whole form, to get you started quickly:

```
var f = new ContactForm()
print(reactHTML(<RenderForm form={f}/>))
/* =>
<div>
  <div><label for="id_subject">Subject:</label> <input maxlength="100" type="text" name="subject" id="id_subject"></div>
  <div><label for="id_message">Message:</label> <input type="text" name="message" id="id_message"></div>
  <div><label for="id_sender">Sender:</label> <input type="email" name="sender" id="id_sender"></div>
  <div><label for="id_ccMyself">Cc myself:</label> <input type="checkbox" name="ccMyself" id="id_ccMyself"></div>
</div>
*/
```

If a form has some user input data, the HTML output will include that data appropriately:

```
var data = {
  subject: 'hello'
, message: 'Hi there'
, sender: 'foo@example.com'
, ccMyself: true
}
var f = new ContactForm({data: data})
print(reactHTML(<RenderForm form={f}/>))
/* =>
<div>
  <div><label for="id_subject">Subject:</label> <input maxlength="100" type="text" name="subject" id="id_subject" value="hello"></div>
  <div><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" value="Hi there"></div>
  <div><label for="id_sender">Sender:</label> <input type="email" name="sender" id="id_sender" value="foo@example.com"></div>
  <div><label for="id_ccMyself">Cc myself:</label> <input type="checkbox" name="ccMyself" id="id_ccMyself" checked=""></div>
</div>
*/
```

- For flexibility, the output does *not* include the a `<form>` or an `<input type="submit">`. It's your job to do that.
- Each field type has a default HTML representation. `CharField` is represented by an `<input type="text">` and `EmailField` by an `<input type="email">`. `BooleanField` is represented by an `<input type="checkbox">`. Note these are merely sensible defaults; you can specify which input to use for a given field by using widgets, which we'll explain shortly.
- The HTML name for each tag is taken directly from its property name in `ContactForm`.
- The text label for each field – e.g. `'Subject:'`, `'Message:'` and `'Cc myself:'` is generated from the field name by splitting on capital letters and lowercasing first letters, converting all underscores to spaces and upper-casing the first letter. Again, note these are merely sensible defaults; you can also specify labels manually.
- Each text label is surrounded in an HTML `<label>` tag, which points to the appropriate form field via its `id`. Its `id`, in turn, is generated by prepending `'id_'` to the field name. The `id` attributes and `<label>` tags are included in the output by default, to follow best practices, but you can change that behavior.

Styling form rows

When defining a Form, there are a few hooks you can use to add `class` attributes to form rows in the default rendering:

- `rowCssClass` – applied to every form row

- `requiredCssClass` – applied to form rows for required fields
- `optionalCssClass` – applied to form rows for optional fields
- `errorCssClass` – applied to form rows for fields which have errors
- `validCssClass` – applied to form rows for fields which have a corresponding value present in `cleanedData`
- `pendingCssClass` – applied to form rows for fields which have a pending asynchronous validation.

To use these hooks, ensure your form has them as prototype or instance properties, e.g. to set them up as prototype properties:

```
var ContactForm = forms.Form.extend({
    rowCssClass: 'row',
    requiredCssClass: 'required',
    optionalCssClass: 'optional',
    errorCssClass: 'error',
    validCssClass: 'valid',
    // ...and the rest of your fields here
})
```

Once you've done that, the generated markup will look something like:

```
var data = {
    subject: 'hello'
, message: 'Hi there'
, sender: ''
, ccMyself: true
}
var f = new ContactForm({data: data})
print(reactHTML(<RenderForm form={f}/>))
/* =>
<div>
  <div class="row valid required"><label for="id_subject">Subject:</label> ...
  <div class="row valid required"><label for="id_message">Message:</label> ...
  <div class="row error required"><label for="id_sender">Sender:</label> ...
  <div class="row valid optional"><label for="id_ccMyself">Cc myself:</label> ...
</div>
*/
```

The `className` string generated for each field when you configure the available CSS properties is also available for use in custom rendering, via a `BoundField`'s `cssClasses()` method.

Configuring form elements' HTML id attributes and `<label>` tags

By default, the form rendering methods include:

- HTML `id` attributes on the form elements.
- The corresponding `<label>` tags around the labels. An HTML `<label>` tag designates which label text is associated with which form element. This small enhancement makes forms more usable and more accessible to assistive devices. It's always a good idea to use `<label>` tags.

The `id` attribute values are generated by prepending `id_` to the form field names. This behavior is configurable, though, if you want to change the `id` convention or remove HTML `id` attributes and `<label>` tags entirely.

Use the `autoId` argument to the `Form` constructor to control the `id` and label behavior. This argument must be `true`, `false` or a string.

If `autoId` is `false`, then the form output will include neither `<label>` tags nor `id` attributes:

```
var f = new ContactForm({autoId: false})
print(reactHTML(<RenderForm form={f}/>))
/* =>
<div>
  <divSubject: <input maxlength="100" type="text" name="subject"></div>
  <divMessage: <input type="text" name="message"></div>
  <divSender: <input type="email" name="sender"></div>
  <divCc myself: <input type="checkbox" name="ccMyself"></div>
</div>
*/
```

If `autoId` is set to `true`, then the form output will include `<label>` tags and will simply use the field name as its `id` for each form field:

```
var f = new ContactForm({autoId: true})
print(reactHTML(<RenderForm form={f}/>))
/* =>
<div>
  <div><label for="subject">Subject:</label> <input maxlength="100" type="text" name="subject" id="subject"></div>
  <div><label for="message">Message:</label> <input type="text" name="message" id="message"></div>
  <div><label for="sender">Sender:</label> <input type="email" name="sender" id="sender"></div>
  <div><label for="ccMyself">Cc myself:</label> <input type="checkbox" name="ccMyself" id="ccMyself"></div>
</div>
*/
```

If `autoId` is set to a string containing a `'{name}'` format placeholder, then the form output will include `<label>` tags, and will generate `id` attributes based on the format string:

```
var f = new ContactForm({autoId: 'id_for_{name}'})
print(reactHTML(<RenderForm form={f}/>))
/* =>
<div>
  <div><label for="id_for_subject">Subject:</label> <input maxlength="100" type="text" name="subject" id="id_for_subject"></div>
  <div><label for="id_for_message">Message:</label> <input type="text" name="message" id="id_for_message"></div>
  <div><label for="id_for_sender">Sender:</label> <input type="email" name="sender" id="id_for_sender"></div>
  <div><label for="id_for_ccMyself">Cc myself:</label> <input type="checkbox" name="ccMyself" id="id_for_ccMyself"></div>
</div>
*/
```

By default, `autoId` is set to the string `'id_{name}'`.

It's possible to customise the suffix character appended to generated labels (default: `' : '`), or omit it entirely, using the `labelSuffix` parameter:

```
var f = new ContactForm({autoId: 'id_for_{name}', labelSuffix: ''})
print(reactHTML(<RenderForm form={f}/>))
/* =>
<div>
  <div><label for="id_for_subject">Subject</label> <input maxlength="100" type="text" name="subject" id="id_for_subject"></div>
  <div><label for="id_for_message">Message</label> <input type="text" name="message" id="id_for_message"></div>
  <div><label for="id_for_sender">Sender</label> <input type="email" name="sender" id="id_for_sender"></div>
  <div><label for="id_for_ccMyself">Cc myself</label> <input type="checkbox" name="ccMyself" id="id_for_ccMyself"></div>
</div>
*/
```

```
f = new ContactForm({autoId: 'id_for_{name}', labelSuffix: ' ->'})
print(reactHTML(<RenderForm form={f}/>))
/* =>
```

```

<div>
  <div><label for="id_for_subject">Subject -&gt;</label> <input maxlength="100" type="text" name="subject" id="id_for_subject"></div>
  <div><label for="id_for_message">Message -&gt;</label> <input type="text" name="message" id="id_for_message"></div>
  <div><label for="id_for_sender">Sender -&gt;</label> <input type="email" name="sender" id="id_for_sender"></div>
  <div><label for="id_for_ccMyself">Cc myself -&gt;</label> <input type="checkbox" name="ccMyself" id="id_for_ccMyself"></div>
</div>
*/

```

Note that the label suffix is added only if the last character of the label isn't a punctuation character.

You can also customise the `labelSuffix` on a per-field basis using the `labelSuffix` argument to `BoundField#labelTag()`.

Notes on field ordering

In the provided default rendering, fields are displayed in the order in which you define them in your form. For example, in the `ContactForm` example, the fields are defined in the order `subject`, `message`, `sender`, `ccMyself`. To reorder the HTML output, just change the order in which those fields are listed in the class.

How errors are displayed

If you render a bound `Form` object, the act of rendering will automatically run the form's validation if it hasn't already happened, and the HTML output will include the validation errors as a `<ul class="errorlist">` near the field:

```

var data = {
  subject: ''
, message: 'Hi there'
, sender: 'invalid email address'
, ccMyself: true
}
var f = new ContactForm({data: data})
print(reactHTML(<RenderForm form={f}/>))
/* =>
<div>
  <div><ul class="errorlist"><li>This field is required.</li></ul><label for="id_subject">Subject:</label> <input type="text" name="subject" id="id_subject"></div>
  <div><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" value="Hi there"></div>
  <div><ul class="errorlist"><li>Enter a valid email address.</li></ul><label for="id_sender">Sender:</label> <input type="email" name="sender" id="id_sender"></div>
  <div><label for="id_ccMyself">Cc myself:</label> <input type="checkbox" name="ccMyself" id="id_ccMyself" checked=""></div>
</div>
*/

```

Customising the error list format

By default, forms use `ErrorList()` to format validation errors. You can pass an alternate constructor for displaying errors at form construction time:

```

var DivErrorList = forms.ErrorList.extend({
  render: function() {
    return <div className="errorlist">
      {this.messages().map(function(error) { return <div>{error}</div> })}
    </div>
  }
})

```

```
f = new ContactForm({data: data, errorConstructor: DivErrorList, autoId: false})
print(reactHTML(<RenderForm form={f}/>))
/* =>
<div>
  <div><div class="errorlist"><div>This field is required.</div></div>Subject: <input maxlength="100"
  <div>Message: <input type="text" name="message" value="Hi there"></div>
  <div><div class="errorlist"><div>Enter a valid email address.</div></div>Sender: <input type="email"
  <div>Cc myself: <input type="checkbox" name="ccMyself" checked></div>
</div>
*/
```

More granular output

To retrieve a single `BoundField()`, use the `Form#boundField()` method on your form, passing the field's name:

```
var form = new ContactForm()
print(reactHTML(form.boundField('subject').render()))
// => <input maxlength="100" type="text" name="subject" id="id_subject">
```

To retrieve all `BoundField` objects, call `Form#boundFields()`:

```
var form = new ContactForm()
form.boundFields().forEach(function(bf) {
  print(reactHTML(bf.render()))
})
/* =>
<input maxlength="100" type="text" name="subject" id="id_subject">
<input type="text" name="message" id="id_message">
<input type="email" name="sender" id="id_sender">
<input type="checkbox" name="ccMyself" id="id_ccMyself">
*/
```

The field-specific output honours the form object's `autoId` setting:

```
var f = new ContactForm({autoId: false})
print(reactHTML(f.boundField('message').render()))
// => <input type="text" name="message">
f = new ContactForm({autoId: 'id_{name}'})
print(reactHTML(f.boundField('message').render()))
// => <input type="text" name="message" id="id_message">
```

`boundField.errors()` returns an object which renders as a `<ul class="errorlist">`:

```
var data = {subject: 'hi', message: '', sender: '', ccMyself: ''}
var f = new ContactForm({data: data, autoId: false})
var bf = f.boundField('message')
print(reactHTML(bf.render()))
// => <input type="text" name="message">
print(bf.errors().messages())
// => ["This field is required."]
print(reactHTML(bf.errors().render()))
// => <ul class="errorlist"><li>This field is required.</li></ul>
bf = f.boundField('subject')
print(bf.errors().messages())
// => []
print(reactHTML(bf.errors().render()))
// =>
```

To separately render the label tag of a form field, you can call its `BoundField#labelTag()` method:

```
var f = new ContactForm()
print(reactHTML(f.boundField('message').labelTag()))
// => <label for="id_message">Message:</label>
```

If you're manually rendering a field, you can access configured CSS classes using the `cssClasses` method:

```
var f = new ContactForm() #
f.requiredCssClass = 'required'
print(f.boundField('message').cssClasses())
// => required
```

Additional classes can be provided as an argument:

```
print(f.boundField('message').cssClasses('foo bar'))
// => foo bar required
```

`boundField.value()` returns the raw value of the field as it would be rendered by a `Widget()`:

```
var initial = {subject: 'welcome'}
var data = {subject: 'hi'}
var unboundForm = new ContactForm({initial: initial})
var boundForm = new ContactForm({data: data, initial: initial})
print(unboundForm.boundField('subject').value())
// => welcome
print(boundForm.boundField('subject').value())
// => hi
```

`boundField.idForLabel()` returns the id of the field. For example, if you are manually constructing a label in JSX:

```
<label htmlFor={form.boundField('myField').idForLabel()}>...</label>
```

Extending forms

When you extend a custom Form, the resulting form will include all fields of its parent form(s), followed by any new fields defined:

```
var ContactFormWithPriority = ContactForm.extend({
  priority: forms.CharField()
})
var f = new ContactFormWithPriority({autoId: false})
print(reactHTML(<RenderForm form={f}/>))
/* =>
<div>
  <div>Subject: <input maxlength="100" type="text" name="subject"></div>
  <div>Message: <input type="text" name="message"></div>
  <div>Sender: <input type="email" name="sender"></div>
  <div>Cc myself: <input type="checkbox" name="ccMyself"></div>
  <div>Priority: <input type="text" name="priority"></div>
</div>
*/
```

Forms can be used as mixins (using Concur's `__mixins__` functionality). In this example, `BeatleForm` mixes in `PersonForm` and `InstrumentForm`, and its field list includes their fields:

```
var PersonForm = forms.Form.extend({
  first_name: forms.CharField(),
```

```
    last_name: forms.CharField()
})
var InstrumentForm = forms.Form.extend({
    instrument: forms.CharField()
})
var BeatleForm = forms.Form.extend({
    __mixins__: [PersonForm, InstrumentForm],
    haircut_type: forms.CharField()
})
var b = new BeatleForm({autoId: false})
print(reactHTML(<RenderForm form={b}/>))
/* =>
<div>
  <div>First name: <input type="text" name="first_name"></div>
  <div>Last name: <input type="text" name="last_name"></div>
  <div>Instrument: <input type="text" name="instrument"></div>
  <div>Haircut type: <input type="text" name="haircut_type"></div>
</div>
*/
```

Prefixes for forms

You can put as many forms as you like inside one `<form>` tag. To give each form its own namespace, use the `prefix` argument:

```
var mother = new PersonForm({prefix: 'mother'})
var father = new PersonForm({prefix: 'father'})
print(reactHTML(<RenderForm form={mother}/>))
/* =>
<div>
  <div><label for="id_mother-first_name">First name:</label> <input type="text" name="mother-first_name">
  <div><label for="id_mother-last_name">Last name:</label> <input type="text" name="mother-last_name">
</div>
*/
print(reactHTML(<RenderForm form={father}/>))
/* =>
<div>
  <div><label for="id_father-first_name">First name:</label> <input type="text" name="father-first_name">
  <div><label for="id_father-last_name">Last name:</label> <input type="text" name="father-last_name">
</div>
*/
```

Client: Working with files

If your browser implements the [File API](#), `form.cleanedData` will contain native `File` objects for any `FileField`, `MultipleFileField` and `ImageField` fields in your form.

While these fields are only currently capable of performing limited validation, having access to `File` objects allows you to more easily implement your own validation based on file size, type and contents at whichever of the available *validation steps* is most appropriate for your needs.

Server: Binding uploaded files to a form

Note: This section deals with a very specific use case: using React to render forms, but performing and subsequently

handling regular form submissions.

This may only be relevant if you're using React purely for server-side rendering, or you're creating an isomorphic app which progressively enhances regular form submissions.

Dealing with forms that have `FileField`, `MultipleFileField` or `ImageField` fields and will be submitted via a regular form POST is a little more complicated than a regular form.

Firstly, in order to upload files, you'll need to make sure that your `<form>` element correctly defines the `enctype` as `"multipart/form-data"`:

```
<form enctype="multipart/form-data" method="POST" action="/foo">
```

Secondly, when you use the form, you need to bind the file data. File data is handled separately to normal form data, so when your form contains a `FileField` and `ImageField`, you will need to specify a `files` argument when creating a form instance. So if we extend our `ContactForm` to include an `ImageField` called `mugshot`, we need to bind the file data containing the `mugshot` image:

```
// Bound form with an image field
var data = {
  subject: 'hello'
, message: 'Hi there'
, sender: 'invalid email address'
, ccMyself: true
}
var fileData = {mugshot: {name: 'face.jpg', size: 123456}}
var f = new ContactFormWithMugshot({data: data, files: fileData})
```

Assuming you're using [Express](#), or a similar library which supports middleware for processing file uploads, you will usually specify `req.files` as the source of file data (just like you'd use `req.body` as the source of form data):

```
// Bound form with an image field, data from the request
var f = new ContactFormWithMugshot({data: req.body, files: req.files})
```

Note: Newforms doesn't really care how you're handling file uploads, just that the object passed as a `file` argument has `FileField` names as its properties and that the corresponding values have `name` and `size` properties.

Constructing an unbound form is the same as always – just omit both form data *and* file data:

```
// Unbound form with a image field
var f = new ContactFormWithMugshot()
```

Testing for multipart forms

If you're writing reusable views or templates, you may not know ahead of time whether your form is a multipart form or not. The `isMultipart()` method tells you whether the form requires multipart encoding for submission:

```
var f = new ContactFormWithMugshot()
print(f.isMultipart())
// => true
```

Here's an example of how you might use this in a React component `render()` method with JSX:

```
<form enctype={form.isMultipart() && 'multipart/form-data'} method="POST" action="/foo">
  ...
</form>
```

3.1.7 Form fields

When you create a new `Form`, the most important part is defining its fields. Each field has custom validation logic, along with a few other hooks.

Although the primary way you'll use a `Field` is in a `Form`, you can also instantiate them and use them directly to get a better idea of how they work. Each `Field` instance has a `clean()` method, which takes a single argument and either throws a `forms.ValidationError` or returns the clean value:

```
var f = forms.EmailField()
print(f.clean('foo@example.com'))
// => foo@example.com
try {
    f.clean('invalid email address')
}
catch (e) {
    print(e.messages())
}
// => ["Enter a valid email address."]
```

Core field arguments

required

By default, each `Field` assumes the value is required, so if you pass an empty value – undefined, null or the empty string ('') – then `clean()` will throw a `ValidationError`.

To specify that a field is *not* required, pass `required: false` to the `Field` constructor:

```
var f = forms.CharField({required: false})
```

If a `Field` has `required: false` and you pass `clean()` an empty value, then `clean()` will return a *normalised* empty value rather than throwing a `ValidationError`. For `CharField`, this will be an empty string. For another `Field` type, it might be null (This varies from field to field.).

label

The `label` argument lets you specify the “human-friendly” label for this field. This is used when the `Field` is displayed in a `Form`.

initial

The `initial` argument lets you specify the initial value to use when rendering this `Field` in an unbound `Form`.

To specify dynamic initial data, see the *initial* option.

widget

The `widget` argument lets you specify a `Widget` to use when rendering this `Field`. You can pass either an instance or a `Widget` constructor. See [Widgets](#) for more information.

helpText

The `helpText` argument lets you specify descriptive text for this Field. If you provide `helpText`, it will be displayed next to the Field when the Field is rendered by one of the default rendering components.

To render raw HTML in help text, specify it using the React convention for raw HTML, which is as an object with an `__html` property.

errorMessages

The `errorMessages` argument lets you override the default messages that the field will throw. Pass in an object with properties matching the error messages you want to override. For example, here is the default error message:

```
var generic = forms.CharField()
try {
    generic.clean('')
}
catch (e) {
    print(e.messages())
}
// => ["This field is required."]
```

And here is a custom error message:

```
var name = forms.CharField({errorMessages: {required: 'Please enter your name.'}})
try {
    name.clean('')
}
catch (e) {
    print(e.messages())
}
// => ["Please enter your name."]
```

The error message codes used by fields are defined below.

validators

The `validators` argument lets you provide a list of additional validation functions for this field.

widgetAttrs

New in version 0.11.

The `widgetAttrs` argument lets you specify additional attributes for the field's Widget without having to specify a new Widget from scratch using the `widget` argument.

This provides a more convenient way to add attributes like `autoFocus`, `className` or `data-` attributes to the field's default Widget **but** has the lowest priority when determining final widget attributes.

For example, you can't override an `Input`-based widget's `type` attribute using `widgetAttrs`, as `type` is set in that widget's `render()` method based on the specific widget constructor being used.

However, `TextInput()` treats the `type` attribute specially, to allow you to override the rendered type when creating an instance:

```
TextInput({attrs: {type: 'tel'}})
```

Or you can extend `TextInput` to create your own widget constructor:

```
var TelInput = TextInput.extend({
    inputType: 'tel'
})
```

Providing choices

Fields and Widgets which take a `choices` argument expect to be given a list containing any of the following:

Choice pairs A choice pair is a list containing exactly 2 elements, which correspond to:

1. the value to be submitted/returned when the choice is selected.
2. the value to be displayed to the user for selection.

For example:

```
var STATE_CHOICES = [
    ['S', 'Scoped']
    , ['D', 'Defined']
    , ['P', 'In-Progress']
    , ['C', 'Completed']
    , ['A', 'Accepted']
]
print(reactHTML(forms.Select().render('state', null, {choices: STATE_CHOICES})))
/* =>
<select name="state">
<option value="S">Scoped</option>
<option value="D">Defined</option>
<option value="P">In-Progress</option>
<option value="C">Completed</option>
<option value="A">Accepted</option>
</select>
*/
```

Grouped lists of choice pairs A list containing exactly 2 elements, which correspond to:

1. A group label
2. A list of choice pairs, as described above

```
var DRINK_CHOICES = [
    ['Cheap', [
        [1, 'White Lightning']
        , [2, 'Buckfast']
        , [3, 'Tesco Gin']
    ]
    , ['Expensive', [
        [4, 'Vieille Bon Secours Ale']
        , [5, 'Château d'Yquem']
        , [6, 'Armand de Brignac Midas']
    ]
    , [7, 'Beer']
]
print(reactHTML(forms.Select().render('drink', null, {choices: DRINK_CHOICES})))
```

```

/* =>
<select name="drink">
<optgroup label="Cheap">
<option value="1">White Lightning</option>
<option value="2">Buckfast</option>
<option value="3">Tesco Gin</option>
</optgroup>
<optgroup label="Expensive">
<option value="4">Vieille Bon Secours Ale</option>
<option value="5">Château d'Yquem</option>
<option value="6">Armand de Brignac Midas</option>
</optgroup>
<option value="7">Beer</option>
</select>
*/

```

As you can see from the 'Beer' example above, grouped pairs can be mixed with ungrouped pairs within the list of choices.

Flat choices New in version 0.5.

If a non-array value is provided where newforms expects to see a choice pair, it will be normalised to a choice pair using the same value for submission and display.

This allows you to pass a flat list of choices when that's all you need:

```

var VOWEL_CHOICES = ['A', 'E', 'I', 'O', 'U']
var f = forms.ChoiceField({choices: VOWEL_CHOICES})
print(f.choices())
// => [['A', 'A'], ['E', 'E'], ['I', 'I'], ['O', 'O'], ['U', 'U']]

var ARBITRARY_CHOICES = [
  ['Numbers', [1, 2,]],
  ['Letters', ['A', 'B']]
]
f.setChoices(ARBITRARY_CHOICES)
print(f.choices())
// => [['Numbers', [[1, 1], [2, 2]]], ['Letters', [['A', 'A'], ['B', 'B']]]

```

Dynamic choices

A common pattern for providing dynamic choices (or indeed, dynamic anything) is to provide your own form constructor and pass in whatever data is required to make changes to `form.fields` as the form is being instantiated.

Newforms provides a `util.makeChoices()` helper function for creating choice pairs from a list of objects using named properties:

```

var ProjectBookingForm = forms.Form.extend({
  project: forms.ChoiceField(),
  hours: forms.DecimalField({minValue: 0, maxValue: 24, maxdigits: 4, decimalPlaces: 2}),
  date: forms.DateField(),

  constructor: function(projects, kwargs) {
    // Call the constructor of whichever form you're extending so that the
    // forms.Form constructor eventually gets called - this.fields doesn't
    // exist until this happens.
    forms.Form.call(this, kwargs)
  }
})

```

```
// Now that this.fields is a thing, make whatever changes you need to -
// in this case, we're going to create a list of pairs of project ids
// and names to set as the project field's choices.
this.fields.project.setChoices(forms.util.makeChoices(projects, 'id', 'name'))
}
})

var projects = [
  {id: 1, name: 'Project 1'}
, {id: 2, name: 'Project 2'}
, {id: 3, name: 'Project 3'}
]
var form = new ProjectBookingForm(projects, {autoId: false})
print(reactHTML((form.boundField('project').render())))
/* =>
<select name="project">
<option value="1">Project 1</option>
<option value="2">Project 2</option>
<option value="3">Project 3</option>
</select>
*/
```

Server-side example of using a form with dynamic choices:

```
// Users are assigned to projects and they're booking time, so we need to:
// 1. Display choices for the projects they're assigned to
// 2. Validate that the submitted project id is one they've been assigned to
var form
var display = function() { res.render('book_time', {form: form}) }
req.user.getProjects(function(err, projects) {
  if (err) { return next(err) }
  if (req.method == 'POST') {
    form = new ProjectBookingForm(projects, {data: req.body})
    if (form.isValid()) {
      return ProjectService.saveHours(user, form.cleanedData, function(err) {
        if (err) { return next(err) }
        return res.redirect('/time/book/')
      })
    }
  }
  else {
    form = new ProjectBookingForm(projects)
  }
  display(form)
})
```

Built-in Field types

newforms comes with a set of Field types that represent common validation needs. This section documents each built-in field.

For each field, we describe the default widget used if you don't specify widget. We also specify the value returned when you provide an empty value (see the section on required above to understand what that means).

Built-in Field type hierarchy

- Field
 - BaseTemporalField
 - * *DateField*
 - * *DateTimeField*
 - * *TimeField*
 - BooleanField
 - * *NullBooleanField*
 - CharField
 - * *EmailField*
 - * *GenericIPAddressField*
 - * *IPAddressField*
 - * *RegexField*
 - * *SlugField*
 - * *URLField*
 - ChoiceField
 - * *FilePathField*
 - * *MultipleChoiceField*
 - *TypedMultipleChoiceField*
 - * *TypedChoiceField*
 - ComboField
 - IntegerField
 - * *DecimalField*
 - * *FloatField*
 - FileField
 - * *ImageField*
 - * *MultipleFileField*
 - MultiValueField
 - * *SplitDateTimeField*

Build-in Fields (A-Z)

BooleanField()

- Default widget: *CheckboxInput* ()
- Empty value: `false`
- Normalises to: A JavaScript `true` or `false` value.

- Validates that the value is `true` (e.g. the check box is checked) if the field has `required: true`.
- Error message keys: `required`

Note: Since all `Field` types have `required: true` by default, the validation condition here is important. If you want to include a boolean in your form that can be either `true` or `false` (e.g. a checked or unchecked checkbox), you must remember to pass in `required: false` when creating the `BooleanField`.

`CharField()`

- Default widget: `TextInput()`
- Empty value: `''` (an empty string)
- Normalises to: A string.
- Validates `maxLength` or `minLength`, if they are provided. Otherwise, all inputs are valid.
- Error message keys: `required`, `maxLength`, `minLength`

Has two optional arguments for validation:

- `maxLength`
- `minLength`

If provided, these arguments ensure that the string is at most or at least the given length.

`ChoiceField()`

- Default widget: `Select()`
- Empty value: `''` (an empty string)
- Normalises to: A string.
- Validates that the given value exists in the list of choices.
- Error message keys: `required`, `invalidChoice`

The `invalidChoice` error message may contain `{value}`, which will be replaced with the selected choice.

Takes one extra argument:

- `choices`

A list of pairs (2 item lists) to use as choices for this field. See [Providing choices](#) for more details.

`TypedChoiceField()`

Just like a `ChoiceField()`, except `TypedChoiceField()` takes two extra arguments, `coerce` and `emptyValue`.

- Default widget: `Select()`
- Empty value: Whatever you've given as `emptyValue`

- Normalises to: A value of the type provided by the `coerce` argument.
- Validates that the given value exists in the list of choices and can be coerced.
- Error message keys: `required`, `invalidChoice`

Takes extra arguments:

- `coerce`

A function that takes one argument and returns a coerced value. Examples include the built-in `Number`, `Boolean` and other types. Defaults to an identity function. Note that coercion happens after input validation, so it is possible to coerce to a value not present in choices.

- `emptyValue`

The value to use to represent “empty.” Defaults to the empty string; `null` is another common choice here. Note that this value will not be coerced by the function given in the `coerce` argument, so choose it accordingly.

`DateField()`

- Default widget: `DateInput()`
- Empty value: `null`
- Normalises to: A JavaScript `Date` object, with its time fields set to zero.
- Validates that the given value is either a `Date`, or string formatted in a particular date format.
- Error message keys: `required`, `invalid`

Takes one optional argument:

- `inputFormats`

A list of `format strings` used to attempt to convert a string to a valid `Date` object.

If no `inputFormats` argument is provided, the default input formats are:

```
[
    '%Y-%m-%d'           // '2006-10-25'
  , '%m/%d/%Y', '%m/%d/%y' // '10/25/2006', '10/25/06'
  , '%b %d %Y', '%b %d, %Y' // 'Oct 25 2006', 'Oct 25, 2006'
  , '%d %b %Y', '%d %b, %Y' // '25 Oct 2006', '25 Oct, 2006'
  , '%B %d %Y', '%B %d, %Y' // 'October 25 2006', 'October 25, 2006'
  , '%d %B %Y', '%d %B, %Y' // '25 October 2006', '25 October, 2006'
]
```

`DateTimeField()`

- Default widget: `DateTimeInput()`
- Empty value: `null`
- Normalises to: A JavaScript `Date` object.
- Validates that the given value is either a `Date` or string formatted in a particular datetime format.
- Error message keys: `required`, `invalid`

Takes one optional argument:

- `inputFormats`

A list of `format strings` used to attempt to convert a string to a valid `Date` object.

If no `inputFormats` argument is provided, the default input formats are:

```
[
    '%Y-%m-%d %H:%M:%S' // '2006-10-25 14:30:59'
  , '%Y-%m-%d %H:%M'    // '2006-10-25 14:30'
  , '%Y-%m-%d'          // '2006-10-25'
  , '%m/%d/%Y %H:%M:%S' // '10/25/2006 14:30:59'
  , '%m/%d/%Y %H:%M'    // '10/25/2006 14:30'
  , '%m/%d/%Y'          // '10/25/2006'
  , '%m/%d/%Y %H:%M:%S' // '10/25/06 14:30:59'
  , '%m/%d/%Y %H:%M'    // '10/25/06 14:30'
  , '%m/%d/%Y'          // '10/25/06'
]
```

`DecimalField()`

- Default widget: `NumberInput()`.
- Empty value: `null`
- Normalises to: A string (since JavaScript doesn't have built-in `Decimal` type).
- Validates that the given value is a decimal. Leading and trailing whitespace is ignored.
- Error message keys: `required`, `invalid`, `maxValue`, `minValue`, `maxDigits`, `maxDecimalPlaces`, `maxWholeDigits`

The `maxValue` and `minValue` error messages may contain `{limitValue}`, which will be substituted by the appropriate limit.

Similarly, the `maxDigits`, `maxDecimalPlaces` and `maxWholeDigits` error messages may contain `{max}`.

Takes four optional arguments:

- `maxValue`
- `minValue`

These control the range of values permitted in the field.

- `maxDigits`

The maximum number of digits (those before the decimal point plus those after the decimal point, with leading zeros stripped) permitted in the value.

- `decimalPlaces`

The maximum number of decimal places permitted.

`EmailField()`

- Default widget: `EmailInput()`
- Empty value: `''` (an empty string)
- Normalises to: A string.

- Validates that the given value is a valid email address, using a moderately complex regular expression.
- Error message keys: `required`, `invalid`

Has two optional arguments for validation, `maxLength` and `minLength`. If provided, these arguments ensure that the string is at most or at least the given length.

`FileField()`

- Default widget: `ClearableFileInput()`
- Empty value: `null`
- Normalises to:
 - Client: a native `File` object, when supported by the browser, otherwise the value of the `<input type="file">`.
 - Server: the given object in `files` - this field just validates what's there and leaves the rest up to you.
- Can validate that non-empty file data has been bound to the form.
- Error message keys: `required`, `invalid`, `missing`, `empty`, `maxLength`

Has two optional arguments for validation, `maxLength` and `allowEmptyFile`. If provided, these ensure that the file name is at most the given length, and that validation will succeed even if the file content is empty.

When you use a `FileField` in a form, you must also remember to *bind the file data to the form*.

The `maxLength` error refers to the length of the filename. In the error message for that key, `{max}` will be replaced with the maximum filename length and `{length}` will be replaced with the current filename length.

`FilePathField()`

- Default widget: `Select()`
- Empty value: `null`
- Normalises to: A string
- Validates that the selected choice exists in the list of choices.
- Error message keys: `required`, `invalidChoice`

The field allows choosing from files inside a certain directory. It takes three extra arguments; only `path` is required:

- `path`

The absolute path to the directory whose contents you want listed. This directory must exist.
- `recursive`

If `false` (the default) only the direct contents of `path` will be offered as choices. If `true`, the directory will be descended into recursively and all descendants will be listed as choices.
- `match`

A regular expression pattern; only files with names matching this expression will be allowed as choices.

- `allowFiles`

Optional. Either `true` or `false`. Default is `true`. Specifies whether files in the specified location should be included. Either this or `allowFolders` must be `true`.

- `allowFolders`

Optional. Either `true` or `false`. Default is `false`. Specifies whether folders in the specified location should be included. Either this or `allowFiles` must be `true`.

`FloatField()`

- Default widget: `NumberInput()`.
- Empty value: `null`
- Normalises to: A JavaScript Number.
- Validates that the given value is a float. Leading and trailing whitespace is allowed.
- Error message keys: `required`, `invalid`, `maxValue`, `minValue`

Takes two optional arguments for validation, `maxValue` and `minValue`. These control the range of values permitted in the field.

`ImageField()`

- Default widget: `ClearableFileInput()`
- Empty value: `null`
- Normalises to: The given object in `files` - this field just validates what's there and leaves the rest up to you.
- Validates that file data has been bound to the form, and that the file is of an image format.
- Error message keys: `required`, `invalid`, `missing`, `empty`, `invalidImage`

Note: Server-side image validation isn't implemented yet.

When you use a `ImageField` in a form, you must also remember to *bind the file data to the form*.

`IntegerField()`

- Default widget: `NumberInput()`.
- Empty value: `null`
- Normalises to: A JavaScript Number.
- Validates that the given value is an integer. Leading and trailing whitespace is allowed.
- Error message keys: `required`, `invalid`, `maxValue`, `minValue`

The `maxValue` and `minValue` error messages may contain `{limitValue}`, which will be substituted by the appropriate limit.

Takes two optional arguments for validation:

- `maxValue`
- `minValue`

These control the range of values permitted in the field.

`IPAddressField()`

Deprecated since version 0.5: This field has been deprecated in favour of `GenericIPAddressField()`.

- Default widget: `TextInput()`
- Empty value: `''` (an empty string)
- Normalises to: A string.
- Validates that the given value is a valid IPv4 address, using a regular expression.
- Error message keys: `required`, `invalid`

`GenericIPAddressField()`

A field containing either an IPv4 or an IPv6 address.

- Default widget: `TextInput()`
- Empty value: `''` (an empty string)
- Normalises to: A string. IPv6 addresses are normalised as described below.
- Validates that the given value is a valid IP address.
- Error message keys: `required`, `invalid`

The IPv6 address normalisation follows [RFC 4291#section-2.2](#) section 2.2, including using the IPv4 format suggested in paragraph 3 of that section, like `::ffff:192.0.2.0`. For example, `2001:0::0:01` would be normalised to `2001::1`, and `::ffff:0a0a:0a0a` to `::ffff:10.10.10.10`. All characters are converted to lowercase.

Takes two optional arguments:

- `protocol`
Limits valid inputs to the specified protocol. Accepted values are `both` (default), `ipv4` or `ipv6`. Matching is case insensitive.
- `unpackIPv4`
Unpacks IPv4 mapped addresses like `::ffff:192.0.2.1`. If this option is enabled that address would be unpacked to `192.0.2.1`. Default is disabled. Can only be used when `protocol` is set to `'both'`.

`MultipleChoiceField()`

- Default widget: `SelectMultiple()`
- Empty value: `[]` (an empty list)
- Normalises to: A list of strings.
- Validates that every value in the given list of values exists in the list of choices.

- Error message keys: `required`, `invalidChoice`, `invalidList`

The `invalidChoice` error message may contain `{value}`, which will be replaced with the selected choice.

Takes one extra required argument, `choices`, as for `ChoiceField`.

`MultipleFileField()`

New in version 0.11: Default widget: `FileInput()` with `multiple` attribute Empty value: `[]` (an empty list) Normalises to:

- – Client: a list of `File` objects, when supported by the browser, otherwise the value of the `<input type="file" multiple>`.
- – Server: the given object in `files` - this field just validates what's there and leaves the rest up to you.
- Can validate that non-empty file data has been bound to the form.
- Error message keys: `required`, `invalid`, `missing`, `empty`, `maxLength`

The `empty` and `maxLength` error messages may contain `{name}`, which will be replaced with the name of the file which failed validation.

Has two optional arguments for validation, `maxLength` and `allowEmptyFile` as for `FileField`.

`TypedMultipleChoiceField()`

Just like a `MultipleChoiceField()`, except `TypedMultipleChoiceField()` takes two extra arguments, `coerce` and `emptyValue`.

- Default widget: `SelectMultiple()`
- Empty value: Whatever you've given as `emptyValue`
- Normalises to: A list of values of the type provided by the `coerce` argument.
- Validates that the given values exists in the list of choices and can be coerced.
- Error message keys: `required`, `invalidChoice`

The `invalidChoice` error message may contain `{value}`, which will be replaced with the selected choice.

Takes two extra arguments, `coerce` and `emptyValue`, as for `TypedChoiceField`.

`NullBooleanField()`

- Default widget: `NullBooleanSelect()`
- Empty value: `null`
- Normalises to: A JavaScript `true`, `false` or `null` value.
- Validates nothing (i.e., it never raises a `ValidationError`).

RegexField()

- Default widget: `TextInput()`
- Empty value: `''` (an empty string)
- Normalises to: A string.
- Validates that the given value matches against a certain regular expression.
- Error message keys: `required`, `invalid`

Takes one required argument:

- `regex`

A regular expression specified either as a string or a compiled regular expression object.

Also takes `maxLength` and `minLength`, which work just as they do for `CharField`.

SlugField()

- Default widget: `TextInput()`
- Empty value: `''` (an empty string)
- Normalises to: A string.
- Validates that the given value contains only letters, numbers, underscores, and hyphens.
- Error messages: `required`, `invalid`

TimeField()

- Default widget: `TextInput()`
- Empty value: `null`
- Normalises to: A JavaScript `Date` object, with its date fields set to 1900-01-01.
- Validates that the given value is either a `Date` or string formatted in a particular time format.
- Error message keys: `required`, `invalid`

Takes one optional argument:

- `inputFormats`

A list of `format strings` used to attempt to convert a string to a valid `Date` object.

If no `inputFormats` argument is provided, the default input formats are:

```
[
    '%H:%M:%S' // '14:30:59'
    , '%H:%M'   // '14:30'
]
```

URLField()

- Default widget: `URLInput()`
- Empty value: `''` (an empty string)
- Normalises to: A string.
- Validates that the given value is a valid URL.
- Error message keys: `required`, `invalid`

Takes the following optional arguments:

- `maxLength`
- `minLength`

These are the same as `CharField.maxLength` and `CharField.minLength`.

Slightly complex built-in Field types**ComboField()**

- Default widget: `TextInput()`
- Empty value: `''` (an empty string)
- Normalises to: A string.
- Validates that the given value against each of the fields specified as an argument to the `ComboField`.
- Error message keys: `required`, `invalid`

Takes one extra argument:

- `fields`

The list of fields that should be used to validate the field's value (in the order in which they are provided):

```
var f = forms.ComboField({fields: [
    forms.CharField({maxLength: 20}),
    forms.EmailField()
]})
print(f.clean('test@example.com'))
// => test@example.com
try {
    f.clean('longemailaddress@example.com')
}
catch (e) {
    print(e.messages())
}
// => ['Ensure this value has at most 20 characters (it has 28).']
```

MultiValueField()

- Default widget: `TextInput()`

- Empty value: '' (an empty string)
- Normalises to: the type returned by the `compress` method of the field.
- Validates that the given value against each of the fields specified as an argument to the `MultiValueField`.
- Error message keys: `required`, `invalid`, `incomplete`

Aggregates the logic of multiple fields that together produce a single value.

This field is abstract and must be extended. In contrast with the single-value fields, fields which extend `js:class:MultiValueField` must not implement `BaseField#clean()` but instead - implement `compress()`.

Takes one extra argument:

- `fields`

A list of fields whose values are cleaned and subsequently combined into a single value. Each value of the field is cleaned by the corresponding field in `fields` – the first value is cleaned by the first field, the second value is cleaned by the second field, etc. Once all fields are cleaned, the list of clean values is combined into a single value by `compress()`.

Also takes one extra optional argument:

- `requireAllFields`

New in version 0.5.

Defaults to `true`, in which case a `required` validation error will be raised if no value is supplied for any field.

When set to `false`, the `Field.required` attribute can be set to `false` for individual fields to make them optional. If no value is supplied for a required field, an `incomplete` validation error will be raised.

A default `incomplete` error message can be defined on the `MultiValueField()`, or different messages can be defined on each individual field. For example:

```
var RegexValidator = forms.validators.RegexValidator
var PhoneField = forms.MultiValueField.extend({
  constructor: function(kwarg) {
    kwarg = kwarg || {}
    // Define one message for all fields
    kwarg.errorMessages = {
      incomplete: 'Enter a country code and phone number.'
    }
    // Or define a different message for each field
    kwarg.fields = [
      forms.CharField({errorMessages: {incomplete: 'Enter a country code.'}, validators:
        RegexValidator({regex: /^\d+$/, message: 'Enter a valid country code.'})
      }},
      forms.CharField({errorMessages: {incomplete: 'Enter a phone number.'}, validators:
        RegexValidator({regex: /^\d+$/, message: 'Enter a valid phone number.'})
      }},
      forms.CharField({required: false, validators: [
        RegexValidator({regex: /^\d+$/, message: 'Enter a valid extension.'})
      ]})
    ]
  }
})
PhoneField.__super__.constructor.call(this, kwarg)
```

```
}  
})
```

- `MultiValueField.widget`

Must extend `MultiWidget()`. Default value is `TextInput()`, which probably is not very useful in this case. Have a nice day :)

- `compress(dataList)`

Takes a list of valid values and returns a “compressed” version of those values – in a single value. For example, `SplitDateTimeField()` is a combines a time field and a date field into a `Date` object.

This method must be implemented in the Field extending `MultiValueField`.

`SplitDateTimeField()`

- Default widget: `SplitDateTimeWidget()`
- Empty value: `null`
- Normalises to: A JavaScript `datetime.datetime` object.
- Validates that the given value is a `datetime.datetime` or string formatted in a particular date-time format.
- Error message keys: `required`, `invalid`, `invalidDate`, `invalidTime`

Takes two optional arguments:

- `inputDateFormats`

A list of **format strings** used to attempt to convert a string to a valid `Date` object with its time fields set to zero.

If no `inputDateFormats` argument is provided, the default input formats for `DateField` are used.

- `inputTimeFormats`

A list of **format strings** used to attempt to convert a string to a valid `Date` object with its date fields set to 1900-01-01.

If no `inputTimeFormats` argument is provided, the default input formats for `TimeField` are used.

Creating custom fields

If the built-in `Field` objects don’t meet your needs, you can easily create custom `Fields`. To do this, just `.extend()` `Field`. Its only requirements are that it implement a `clean()` method and that its constructor() accepts the core arguments mentioned above (`required`, `label`, `initial`, `widget`, `helpText`) in an argument object.

3.1.8 Form and Field validation

Form validation happens when the data is cleaned. If you want to customise this process, there are various places you can change, each one serving a different purpose. Three types of cleaning methods are run during form processing.

These are normally executed when you call the `validate()` method on a form or you interact with a field when the form is using event-based validation.

In general, any cleaning method can throw a `ValidationError` if there is a problem with the data it is processing, passing the relevant information to the `ValidationError` constructor.

Most validation can be done using *validators* – helpers that can be reused easily. Validators are functions that take a single argument and throw a `ValidationError` on invalid input. Validators are run after the field's `toJavaScript()` and `validate()` methods have been called.

Validation steps and order

Validation of a Form is split into several steps, which can be customised or overridden:

- The `toJavaScript()` method on a Field is the first step in every validation. It coerces the value to the correct datatype and throws a `ValidationError` if that is not possible. This method accepts the raw value from the widget and returns the converted value. For example, a `FloatField` will turn the data into a JavaScript Number or throw a `ValidationError`.
- The `validate()` method on a Field handles field-specific validation that is not suitable for a validator. It takes a value that has been coerced to the correct datatype and throws a `ValidationError` on any error.

This method does not return anything and shouldn't alter the value. You should override it to handle validation logic that you can't or don't want to put in a validator.
- The `runValidators()` method on a Field runs all of the field's validators and aggregates all the errors into a single `ValidationError`. You shouldn't need to override this method.
- The `clean()` method on a Field. This is responsible for running `toJavaScript`, `validate` and `runValidators` in the correct order and propagating their errors. If, at any time, any of the methods throws a `ValidationError`, the validation stops and that error is thrown. This method returns the clean data, which is then inserted into the `cleanedData` object of the form.
- Field-specific cleaning/validation hooks on the Form. If your form includes a `clean<FieldName>()` (or `clean_<fieldName>()`) method in its definition, it will be called for the field its name matches. This method is not passed its field's data as an argument. You will need to look up the value of the field in `this.cleanedData` (it will be in `cleanedData` because the general field `clean()` method, above, has already cleaned the data once).

For example, if you wanted to validate that the content of a `CharField` called `serialNumber` was unique, implementing `cleanSerialNumber()` would provide the right place to do this.

- The Form `clean()` method. This method can perform any validation that requires access to multiple fields from the form at once. This is where you would perform checks like password or email confirmation fields being equal to the original input.

Since the field validation methods have been run by the time `clean()` is called, you also have access to the form's `errors()`, which contains all the errors thrown by cleaning of individual fields.

Note that any errors thrown by your `form.clean()` override will not be associated with any field in particular. They go into a special "field" (called `__all__`), which you can access via the `nonFieldErrors()` method if you need to. If you want to attach errors to a specific field in the form, you need to call `form.addError()`.

These methods are run in the order given above, one field at a time. That is, for each field in the form (in the order they are declared in the form definition), the `field.clean()` method (or its override) is run, then `clean<Fieldname>()` (or `clean_<fieldName>()`) if defined. Finally, the `form.clean()` method, or its override, is executed whether or not the previous methods have thrown errors.

Examples of each of these methods are provided below.

As mentioned, any of these methods can throw a `ValidationError`. For any field, if the `field.clean()` method throws a `ValidationError`, any field-specific cleaning method is not called. However, the cleaning methods for all remaining fields are still executed.

Throwing `ValidationError`

In order to make error messages flexible and easy to override, consider the following guidelines:

- Provide a descriptive error code to the constructor when possible:

```
forms.ValidationError('Invalid value', {code: 'invalid'})
```

- Don't coerce variables into the message; use placeholders and the `params` argument of the constructor:

```
forms.ValidationError('Invalid value: {value}', {params: {value: '42'}})
```

Putting it all together:

```
throw forms.ValidationError('Invalid value: {value}', {
    code: 'invalid',
    params: {value: '42'}
})
```

Following these guidelines is particularly useful to others if you write reusable forms and form fields.

If you're at the end of the validation chain (i.e. your form's `clean()`) and you know you will *never* need to override your error message (or even just... *because*) you can still opt for the less verbose:

```
forms.ValidationError('Invalid value: ' + value)
```

Throwing multiple errors

If you detect multiple errors during a cleaning method and wish to signal all of them to the form submitter, it is possible to pass a list of errors to the `ValidationError` constructor.

It's recommended to pass a list of `ValidationError` instances with codes and `params` but a list of strings will also work:

```
throw forms.ValidationError([
    forms.ValidationError('Error 1', {code: 'error1'}),
    forms.ValidationError('Error 2', {code: 'error2'})
])

throw forms.ValidationError(['Error 1', 'Error 2'])
```

Using validation in practice

The previous sections explained how validation works in general for forms. Since it can sometimes be easier to put things into place by seeing each feature in use, here are a series of small examples that use each of the previous features.

Using validators

Fields support use of utility functions known as validators. A validator is a function that takes a value and returns nothing if the value is valid, or throws a `ValidationError()` if not. These can be passed to a field's constructor, via the field's `validators` argument, or defined on the field's prototype as a `defaultValidators` property.

Let's have a look at a basic implementation of newforms' `SlugField`:

```
var MySlugField = forms.CharField.extend({
    defaultValidators: [forms.validators.validateSlug]
})
```

As you can see, a basic `SlugField` is just a `CharField` with a customised validator that validates that submitted text obeys some character usage rules. This can also be done on field definition so:

```
var field = new MySlugField()
```

is equivalent to:

```
var field = forms.CharField({validators: [forms.validators.validateSlug]})
```

Common cases such as validating against an email or a regular expression can be handled using existing validators available in newforms. For example, `validateSlug()` is a function created by passing a slug-matching `RegExp` to the `RegexValidator()` function factory.

Form field default cleaning

Let's firstly create a custom form field that validates its input is a string containing comma-separated email addresses:

```
var MultiEmailField = forms.Field.extend({
    /**
     * Normalise data to a list of strings.
     */
    toJavaScript: function(value) {
        // Return an empty list if no input was given
        if (this.isEmptyValue(value)) {
            return []
        }
        return value.split(/, ?/g)
    },

    /**
     * Check if value consists only of valid emails.
     */
    validate: function(value) {
        // Use the parent's handling of required fields, etc.
        MultiEmailField.__super__.validate.call(this, value)
        value.map(forms.validators.validateEmail)
    }
})
```

Let's create a simple `ContactForm` to demonstrate how you'd use this field:

```
var ContactForm = forms.Form.extend({
    subject: forms.CharField({maxLength: 100}),
    message: forms.CharField(),
    sender: forms.EmailField(),
    recipients: new MultiEmailField(),
    ccMyself: forms.BooleanField({required: false})
})
```

Cleaning a specific field

Suppose that in our `ContactForm`, we want to make sure that the `recipients` field always contains the address `"fred@example.com"`. This is validation that is specific to our form, so we don't want to put it into the general `MultiEmailField`. Instead, we write a cleaning function that operates on the `recipients` field, like so:

```
var ContactForm = forms.Form.extend({
  // Everything as before
  // ...

  cleanRecipients: function() {
    var recipients = this.cleanedData.recipients
    if (recipients.indexOf('fred@example.com') == -1) {
      throw forms.ValidationError('You forgot about Fred!')
    }
  }
})
```

Changed in version 0.10: You can no longer return a value from a custom field cleaning method to update the field's `cleanedData`.

Cleaning and validating fields that depend on each other

form.clean() There are two ways to report any errors from this step. Probably the most common method is to display the error at the top of the form. To create such an error, you can throw a `ValidationError` from the `clean()` method. For example:

```
var ContactForm = forms.Form.extend({
  // Everything as before
  // ...

  clean: function() {
    var ccMyself = this.cleanedData.ccMyself
    var subject = this.cleanedData.subject

    if (ccMyself && subject) {
      // Only do something if both fields are valid so far
      if (subject.indexOf('help') == -1) {
        throw forms.ValidationError(
          "Did not send for 'help' in the subject despite CC'ing yourself."
        )
      }
    }
  }
})
```

Another approach might involve assigning the error message to one of the fields. In this case, let's assign an error message to both the "subject" and "ccMyself" rows in the form display:

```
var ContactForm = forms.Form.extend({
  // Everything as before
  // ...

  clean: function() {
    var cleanedData = ContactForm.__super__.clean.call(this)
    var ccMyself = this.cleanedData.ccMyself
    var subject = this.cleanedData.subject
```

```

    if (ccMyself && subject && subject.indexOf('help') == -1) {
        var message = "Must put 'help' in subject when cc'ing yourself."
        this.addError('ccMyself', message)
        this.addError('subject', message)
    }
}
}

```

The second argument to `addError()` can be a simple string, or preferably an instance of `ValidationError`. See [Throwing ValidationError](#) for more details. Note that `addError()` automatically removes the field from `cleanedData`.

Specifying fields used in cross-field validation New in version 0.9.

To let a form know which fields are used in cross-field validation, specify its `clean()` method as an array of field named followed by the cleaning function itself.

In scenarios where the form is being partially updated, such as when individual field input values are being updated and validated when an `onChange` event fires, if this information is available cross-field cleaning will only be performed if one of the fields it uses is affected.

```

var PersonForm = forms.Form.extend({
    firstName: forms.CharField({required: false, maxLength: 50}),
    lastName: forms.CharField({required: false, maxLength: 50}),
    jobTitle: forms.CharField({required: false, maxLength: 100}),
    organisation : forms.CharField({required: false}),

    clean: ['firstName', 'lastName', function() {
        if (!this.cleanedData.firstName && !this.cleanedData.lastName) {
            throw forms.ValidationError('A first name or last name is required.')
        }
    }]
})

```

Asynchronous validation

New in version 0.10.

For some validation you may need to access an external data source, such as a web service, database or filesystem. In JavaScript, these tend to be asynchronous operations.

You can let newforms know that a custom field – or cross-field `clean()` – validation method will be async by defining it with a single parameter in its function signature. It doesn't matter what this is called, but it's conventionally called `callback` or `cb`:

```

cleanUsername: function(callback) {
    // ...
}

```

When your custom cleaning method is finished whatever async operation it needs to perform, it *must* call the callback function to let the form know it can finish the validation process.

The callback has the following signature:

```

function callback(error, validationError)

```

- `error` – an `Error` indicating that something went wrong with the async operation. Any falsy value can be passed if there was no error, but it's conventional to pass `null` in that case. This style of error reporting is known as an “errback”.
- `validationError` – an error message if the field's value was invalid, this can be a simple string or a `ValidationError`.

If async validation determines that the input is valid, you must still call the callback to let newforms know you're done. It can be called without any arguments in this case.

The callback must only be called once, so take care with your custom validation logic, branching or returning early as necessary to avoid calling it multiple times.

Async field validation example

A common use case for async validation is checking if a username is available in a signup form:

```
cleanUsername: function(callback) {
  http.post('/checkuser', {username: this.cleanedData.username}, function(err, res) {
    // There was an error during the HTTP request
    if (err) {
      return callback(err)
    }

    // The username is already taken
    if (res.alreadyTaken) {
      return callback(null, forms.ValidationError('This username is already taken.'))
    }

    // The username is available
    callback()
  })
}
```

In this live example, someone has registered every possible username containing a vowel from the English alphabet, which will randomly take between 1 and 2 seconds to validate:

Cancelling async validation

Asynchronous validation will be cancelled if the user is able to make a change which re-triggers validation while their last change is still being validated.

From the newforms side of things, this effectively involves ignoring calls to the callback which was handed to async validation function.

Tip: In order to support cancelling asynchronous validation, you should always call the given callback with validation results, rather than modifying the form directly via `this`. This allows newforms to ignore your eventual callback if the data is was validating is stale.

If an async validation involves a potentially long-running or (in some way) expensive operation which you'd like to cancel should this happen, you can give newforms a callback to call by returning an object with an `onCancel()` function, after your async validation has started:

```
cleanExpensiveField: function(callback) {
  var request = http.post(
    // ...
```



```

)

return {
    onCancel: function() {
        request.cancel()
    }
}
}

```

Combining sync and async validation

If you have custom validation which can be performed locally, as well as async validation, you can combine the two in the same custom cleaning method.

If validation fails before it reaches the asynchronous part, throwing a `ValidationError` or explicitly returning `false` will let newforms know that asynchronous validation was never initiated:

```

cleanUsername: function(callback) {
    var username = this.cleanedData.username

    // Throwing a ValidationError skips remaining validation for this field
    if (username != esrever.reverse(username)) {
        throw forms.ValidationError('Usernames must be palindromes.')
    }

    // Returning false explicitly lets newforms know not to wait for a callback
    if (/[aeiou].test(username)) {
        this.addError('useranme', 'Usernames must not contain vowels.')
        return false
    }

    http.post(
        // ...
    )
}

```

3.1.9 Widgets

A widget is a representation of an HTML input element. The widget handles the rendering of the HTML, and the extraction of data from a data object that corresponds to how the widget's values(s) would be submitted by a form.

Tip: Widgets should not be confused with the [form fields](#). Form fields deal with the logic of input validation and are used directly in templates. Widgets deal with rendering of HTML form input elements on the web page and extraction of raw submitted data. However, widgets do need to be *assigned* to form fields.

Specifying widgets

Whenever you specify a field on a form, newforms will use a default widget that is appropriate to the type of data that is to be displayed. To find which widget is used on which field, see the documentation about [Build-in Field types](#).

However, if you want to use a different widget for a field, you can just use the `widget` argument on the field definition. For example:

```
var CommentForm = forms.Form.extend({
    name: forms.CharField(),
    url: forms.URLField(),
    comment: forms.CharField({widget: forms.Textarea})
})
```

This would specify a form with a comment that uses a larger `Textarea()` widget, rather than the default `TextInput()` widget.

Setting arguments for widgets

Many widgets have optional extra arguments; they can be set when defining the widget on the field. In the following example, we set additional HTML attributes to be added to the `TextArea` to control its display:

```
var CommentForm = forms.Form.extend({
    name: forms.CharField(),
    url: forms.URLField(),
    comment: forms.CharField({
        widget: forms.Textarea({attrs: {rows: 6, cols: 60}})
    })
})
```

See the [Built-in widgets](#) for more information about which widgets are available and which arguments they accept.

Customising widget attributes

New in version 0.11.

Sometimes you just need to add some extra attributes to a field's default widget. Instead of completely redefining the widget as shown above, you can provide extra attributes using the field's `widgetAttrs` argument. For example, if we want focus to be given to the name field when the form is first rendered:

```
var CommentForm = forms.Form.extend({
    name: forms.CharField({widgetAttrs: {autoFocus: true}}),
    url: forms.URLField(),
    comment: forms.CharField({widget: forms.Textarea})
})
```

Widgets inheriting from the Select widget

Widgets inheriting from the `Select()` widget deal with choices. They present the user with a list of options to choose from. The different widgets present this choice differently; the `Select()` widget itself uses a `<select>` HTML list representation, while `RadioSelect()` uses radio buttons.

`Select()` widgets are used by default on `ChoiceField()` fields. The choices displayed on the widget are inherited from the `ChoiceField()` and setting new choices with `ChoiceField#setChoices()` will update `Select.choices`. For example:

```
var CHOICES = [['1', 'First'], ['2', 'Second']]
var field = forms.ChoiceField({choices: CHOICES, widget: forms.RadioSelect})
print(field.choices())
// => [['1', 'First'], ['2', 'Second']]
print(field.widget.choices)
// => [['1', 'First'], ['2', 'Second']]
field.widget.choices = []
```

```
field.setChoices(['1', 'First and only'])
print(field.widget.choices)
// => [['1', 'First and only']]
```

Widgets which offer a `choices` property can however be used with fields which are not based on choice – such as a `CharField()` – but it is recommended to use a `ChoiceField()`-based field when the choices are inherent to the model and not just the representational widget.

Customising widget instances

Widgets are rendered with minimal markup - by default there are no CSS class names applied, or any other widget-specific attributes. This means, for example, that all `TextInput()` widgets will appear the same on your pages.

Styling widget instances

If you want to make one widget instance look different from another, you will need to specify additional attributes at the time when the widget object is instantiated and assigned to a form field (and perhaps add some rules to your CSS files).

For example, take the following simple form:

```
var CommentForm = forms.Form.extend({
  name: forms.CharField(),
  url: forms.URLField(),
  comment: forms.CharField()
})
```

This form will include three default `TextInput()` widgets, with default rendering – no CSS class, no extra attributes. This means that the input boxes provided for each widget will be rendered exactly the same:

```
var f = new CommentForm({autoId: false})
print(reactHTML(<RenderForm form={f}/>))
/* =>
<div>
  <div>Name: <input type="text" name="name"></div>
  <div>Url: <input type="url" name="url"></div>
  <div>Comment: <input type="text" name="comment"></div>
</div>
*/
```

On a real Web page, you probably don't want every widget to look the same. You might want a larger input element for the comment, and you might want the 'name' widget to have some special CSS class. It is also possible to specify the 'type' attribute to take advantage of the new HTML5 input types. To do this, you use the `Widget.attrs` argument when creating the widget:

```
var CommentForm = forms.Form.extend({
  name: forms.CharField({
    widget: forms.TextInput({attrs: {className: 'special'}})
  }),
  url: forms.URLField(),
  comment: forms.CharField({widget: forms.TextInput({attrs: {size: '40'}})})
})
```

Note: Widgets are rendered as `ReactElement` objects – in the example above, we used `className` instead of `class` as React has standardised on the JavaScript-safe versions of attribute names, which avoid conflicting with JavaScript reserved words.

The extra attributes will then be included in the rendered output:

```
var f = new CommentForm({autoId: false})
print(reactHTML(<RenderForm form={f}/>))
/* =>
<div>
  <div>Name: <input class="special" type="text" name="name"></div>
  <div>Url: <input type="url" name="url"></div>
  <div>Comment: <input size="40" type="text" name="comment"></div>
</div>
*/
```

You can also set the HTML id using `Widget.attrs`.

Base Widgets

Base widgets `Widget()` and `MultiWidget()` are extended by all the *built-in widgets* and may serve as a foundation for custom widgets.

Widget()

This abstract widget cannot be rendered, but provides the basic attribute `Widget.attrs`. You may also implement or override the `render()` method on custom widgets.

`widget.attrs` An object containing HTML attributes to be set on the rendered widget:

```
var name = forms.TextInput({attrs: {size:10, title: 'Your name'}})
print(reactHTML(name.render('name', 'A name'))
// => <input size="10" title="Your name" type="text" name="name" value="A name">
```

Key Widget methods are:

`Widget#render()` Returns a `ReactElement` representation of the widget. This method must be implemented by extending widgets, or an `Error` will be thrown.

The ‘value’ given is not guaranteed to be valid input, therefore extending widgets should program defensively.

`Widget#valueFromData()` Given an object containing input data and this widget’s name, returns the value of this widget. Returns `null` if a value wasn’t provided.

MultiWidget()

A widget that is composed of multiple widgets. `MultiWidget()` works hand in hand with the `MultiValueField()`.

`MultiWidget` has one required argument:

`MultiWidget.widgets` A list containing the widgets needed.

And one required method:

`MultiWidget#decompress()` This method takes a single “compressed” value from the field and returns a list of “decompressed” values. The input value can be assumed valid, but not necessarily non-empty.

This method **must be implemented** by the widgets extending `MultiWidget`, and since the value may be empty, the implementation must be defensive.

The rationale behind “decompression” is that it is necessary to “split” the combined value of the form field into the values for each widget.

An example of this is how `SplitDateTimeWidget()` turns a `Date` value into a list with date and time split into two separate values.

Tip: Note that `MultiValueField()` has a complementary method `MultiValueField#compress()` with the opposite responsibility - to combine cleaned values of all member fields into one.

Other methods that may be useful to implement include:

`MultiWidget#render()` The `value` argument must be handled differently in this method than in `Widget#render()` because it has to figure out how to split a single value for display in multiple widgets.

The `value` argument used when rendering can be one of two things:

- A list.
- A single value (e.g., a string) that is the “compressed” representation of a list of values.

If `value` is a list, the output of `MultiWidget#render()` will be a concatenation of rendered child widgets. If `value` is not a list, it will first be processed by the method `MultiWidget#decompress()` to create the list and then rendered.

When `render()` runs, each value in the list is rendered with the corresponding widget – the first value is rendered in the first widget, the second value is rendered in the second widget, etc.

Unlike in the single value widgets, `render()` doesn’t have to be implemented by extending widgets.

`MultiWidget#formatOutput()` Given a list of rendered widgets (as `ReactElement` objects), returns the list or a `ReactElement` object containing the widgets. This hook allows you to lay out the widgets any way you’d like.

Here’s an example widget which extends `MultiWidget()` to display a date with the day, month, and year in different select boxes. This widget is intended to be used with a `DateField()` rather than a `MultiValueField()`, so we’ve implemented `Widget#valueFromData()`:

```
var DateSelectorWidget = forms.MultiWidget.extend({
  constructor: function(kwargs) {
    kwargs = extend({attrs: {}}, kwargs)
    widgets = [
      forms.Select({choices: range(1, 32), attrs: kwargs.attrs}),
      forms.Select({choices: range(1, 13), attrs: kwargs.attrs}),
      forms.Select({choices: range(2012, 2017), attrs: kwargs.attrs})
    ]
    forms.MultiWidget.call(this, widgets, kwargs)
  },

  decompress: function(value) {
    if (value instanceof Date) {
      return [value.getDate(),
              value.getMonth() + 1, // Make month 1-based for display
              value.getFullYear()]
    }
    return [null, null, null]
  }
})
```

```
},  
  
formatOutput: function(renderedWidgets) {  
    return React.createElement('div', null, renderedWidgets)  
},  
  
valueFromData: function(data, files, name) {  
    var parts = this.widgets.map(function(widget, i) {  
        return widget.valueFromData(data, files, name + '_' + i)  
    })  
    parts.reverse() // [d, m, y] => [y, m, d]  
    return parts.join('-')  
}  
})
```

The constructor creates several `Select()` widgets in a list. The “super” constructor uses this list to setup the widget.

The `MultiWidget#formatOutput()` method is fairly vanilla here (in fact, it’s the same as what’s been implemented as the default for `MultiWidget`), but the idea is that you could add custom HTML between the widgets should you wish.

The required method `MultiWidget#decompress()` breaks up a `Date` value into the day, month, and year values corresponding to each widget. Note how the method handles the case where `value` is `null`.

The default implementation of `Widget#valueFromData()` returns a list of values corresponding to each `Widget`. This is appropriate when using a `MultiWidget` with a `MultiValueField()`, but since we want to use this widget with a `DateField()` which takes a single value, we have overridden this method to combine the data of all the subwidgets into a ‘yyyy-mm-dd’ formatted date string and returns it for validation by the `DateField()`.

Built-in widgets

Newforms provides a representation of all the basic HTML widgets, plus some commonly used groups of widgets, including *the input of text, various checkboxes and selectors, uploading files, and handling of multi-valued input*.

Widgets handling input of text

These widgets make use of the HTML elements `<input>` and `<textarea>`.

`TextInput()`

Text input: `<input type="text" ...>`

`NumberInput()`

Text input: `<input type="number" ...>`

`EmailInput()`

Text input: `<input type="email" ...>`

URLInput ()

Text input: `<input type="url" ...>`

PasswordInput ()

Password input: `<input type='password' ...>`

Takes one optional argument:

- `PasswordInput.renderValue`
Determines whether the widget will have a value filled in when the form is re-displayed after a validation error (default is `false`).

Textarea ()

Text area: `<textarea>...</textarea>`

HiddenInput ()

Hidden input: `<input type='hidden' ...>`

Note that there also is a `MultipleHiddenInput ()` widget that encapsulates a set of hidden input elements.

DateInput ()

Date input as a simple text box: `<input type='text' ...>`

Takes same arguments as `TextInput ()`, with one more optional argument:

- `DateInput.format`
The format in which this field's initial value will be displayed.

If no `format` argument is provided, the default format is the first format found in the current locale's `DATE_INPUT_FORMATS`.

DateTimeInput ()

Date/time input as a simple text box: `<input type='text' ...>`

Takes same arguments as `TextInput ()`, with one more optional argument:

- `DateTimeInput.format`
The format in which this field's initial value will be displayed.

If no `format` argument is provided, the default format is the first format found in the current locale's `DATETIME_INPUT_FORMATS`.

TimeInput ()

Time input as a simple text box: `<input type='text' ...>`

Takes same arguments as `TextInput ()`, with one more optional argument:

- `TimeInput.format`

The format in which this field's initial value will be displayed.

If no `format` argument is provided, the default format is the first format found in the current locale's `TIME_INPUT_FORMATS`.

Selector and checkbox widgets**CheckboxInput ()**

Checkbox: `<input type='checkbox' ...>`

Takes one optional argument:

- `CheckboxInput.checkTest`

A function that takes the value of the `CheckBoxInput` and returns `true` if the checkbox should be checked for that value.

Select ()

Select widget: `<select><option ...>...</select>`

- `Select.choices`

This attribute is optional when the form field does not have a `choices` attribute. If it does, it will override anything you set here when the attribute is updated on the `Field()`.

NullBooleanSelect ()

Select widget with options 'Unknown', 'Yes' and 'No'

SelectMultiple ()

Similar to `Select`, but allows multiple selection: `<select multiple='multiple'>...</select>`

RadioSelect ()

Similar to `Select`, but rendered as a list of radio buttons within `` tags:

```
<ul>
  <li><input type='radio' ...></li>
  ...
</ul>
```


For more granular control over the generated markup, you can loop over the radio buttons. Assuming a form `myForm` with a field `beatles` that uses a `RadioSelect` as its widget:

```
myForm.boundField('beatles').subWidgets().map(function(radio) {
    return <div className="myRadio">{radio.render()}</div>
})
```

This would generate the following HTML:

```
<div class="myRadio">
  <label for="id_beatles_0"><input id="id_beatles_0" type="radio" name="beatles" value="john"> J
</div>
<div class="myRadio">
  <label for="id_beatles_1"><input id="id_beatles_1" type="radio" name="beatles" value="paul"> P
</div>
<div class="myRadio">
  <label for="id_beatles_2"><input id="id_beatles_2" type="radio" name="beatles" value="george">
</div>
<div class="myRadio">
  <label for="id_beatles_3"><input id="id_beatles_3" type="radio" name="beatles" value="ringo">
</div>
```

That included the `<label>` tags. To get more granular, you can use each radio button's `tag()`, `choiceLabel` and `idForLabel()`. For example, this code...:

```
myForm.boundField('beatles').subWidgets().map(function(radio) {
    return <label htmlFor={radio.idForLabel()}>
      {radio.choiceLabel}
      <span className="radio">{radio.tag()}</span>
    </label>
  })
```

...will result in the following HTML:

```
<label for="id_beatles_0">
  John
  <span class="radio"><input id="id_beatles_0" type="radio" name="beatles" value="john"></span>
</label>
<label for="id_beatles_1">
  Paul
  <span class="radio"><input id="id_beatles_1" type="radio" name="beatles" value="paul"></span>
</label>
<label for="id_beatles_2">
  George
  <span class="radio"><input id="id_beatles_2" type="radio" name="beatles" value="george"></span>
</label>
<label for="id_beatles_3">
  Ringo
  <span class="radio"><input id="id_beatles_3" type="radio" name="beatles" value="ringo"></span>
</label>
```

If you decide not to loop over the radio buttons – e.g., if your layout simply renders the `beatles` `BoundField` – they'll be output in a `` with `` tags, as above.

CheckboxSelectMultiple()

Similar to `SelectMultiple()`, but rendered as a list of check buttons:

```
<ul>
  <li><input type='checkbox' ...></li>
  ...
</ul>
```

Like *RadioSelect()*, you can loop over the individual checkboxes making up the lists.

File upload widgets

`FileInput()`

File upload input: `<input type='file' ...>`

`ClearableFileInput()`

File upload input: `<input type='file' ...>`, with an additional checkbox input to clear the field's value, if the field is not required and has initial data.

Composite widgets

`MultipleHiddenInput()`

Multiple `<input type='hidden' ...>` widgets.

A widget that handles multiple hidden widgets for fields that have a list of values.

- `MultipleHiddenInput.choices`

This attribute is optional when the form field does not have a `choices` attribute. If it does, it will override anything you set here when the attribute is updated on the *Field()*.

`SplitDateTimeWidget()`

Wrapper (using *MultiWidget()*) around two widgets: *DateInput()* for the date, and *TimeInput()* for the time.

`SplitDateTimeWidget` has two optional attributes:

- `SplitDateTimeWidget.dateFormat`

Similar to `DateInput.format`

- `SplitDateTimeWidget.timeFormat`

Similar to `TimeInput.format`

`SplitHiddenDateTimeWidget()`

Similar to *SplitDateTimeWidget()*, but uses *HiddenInput()* for both date and time.

3.1.10 Formsets

A formset is a layer of abstraction to work with multiple forms on the same page. It can be best compared to a data grid. Let's say you have the following form:

```
var ArticleForm = forms.Form.extend({
    title: forms.CharField(),
    pubDate: forms.DateField()
})
```

You might want to allow the user to create several articles at once. To create a formset out of an `ArticleForm`, use `FormSet.extend()`:

```
var ArticleFormSet = forms.FormSet.extend({form: ArticleForm})
```

You have now created a formset named `ArticleFormSet`. The formset gives you the ability to iterate over the forms within it and display them as you would any other form:

```
var formset = new ArticleFormSet()
formset.forms().forEach(function(form) {
    print(reactHTML(<RenderForm form={form}/>))
})
/* =>
<div>
  <div><label for="id_form-0-title">Title:</label> <input type="text" name="form-0-title" id="id_form-0-title">
  <div><label for="id_form-0-pubDate">Pub date:</label> <input type="text" name="form-0-pubDate" id="id_form-0-pubDate">
</div>
</div>
*/
```

As you can see it only displayed one empty form. The number of empty forms displayed is controlled by the `extra` parameter. By default, `FormSet()` defines one extra form; the following example will display two blank forms:

```
var ArticleFormSet = forms.FormSet.extend({form: ArticleForm, extra: 2})
```

Using initial data with a formset

Initial data is what drives the main usability of a formset. As shown above you can define the number of extra forms. What this means is that you are telling the formset how many forms to show in addition to the number of forms it generates from the initial data. Let's take a look at an example:

```
var ArticleFormSet = forms.FormSet.extend({form: ArticleForm, extra: 2})
var formset = new ArticleFormSet({initial: [
    {title: "Django's docs are open source!", pubDate: new Date()}
]})
formset.forms().forEach(function(form) {
    print(reactHTML(<RenderForm form={form}/>))
})
/* =>
<div>
  <div><label for="id_form-0-title">Title:</label> <input type="text" name="form-0-title" id="id_form-0-title">
  <div><label for="id_form-0-pubDate">Pub date:</label> <input type="text" name="form-0-pubDate" id="id_form-0-pubDate">
</div>
</div>
<div>
  <div><label for="id_form-1-title">Title:</label> <input type="text" name="form-1-title" id="id_form-1-title">
  <div><label for="id_form-1-pubDate">Pub date:</label> <input type="text" name="form-1-pubDate" id="id_form-1-pubDate">
</div>
</div>
<div>
  <div><label for="id_form-2-title">Title:</label> <input type="text" name="form-2-title" id="id_form-2-title">
  <div><label for="id_form-2-pubDate">Pub date:</label> <input type="text" name="form-2-pubDate" id="id_form-2-pubDate">
</div>
</div>
*/
```

```
<div><label for="id_form-2-pubDate">Pub date:</label> <input type="text" name="form-2-pubDate" id=
</div>
*/
```

There are now a total of three forms showing above. One for the initial data that was passed in and two extra forms. Also note that we are passing in a list of objects as the initial data.

Limiting the maximum number of forms

The `maxNum` parameter to `FormSet()` gives you the ability to limit the maximum number of empty forms the formset will display:

```
var ArticleFormSet = forms.FormSet.extend({
  form: ArticleForm,
  extra: 2,
  maxNum: 1
})
var formset = new ArticleFormSet()
formset.forms().forEach(function(form) {
  print(reactHTML(<RenderForm form={form}/>))
})
/* =>
<div>
  <div><label for="id_form-0-title">Title:</label> <input type="text" name="form-0-title" id="id_for
  <div><label for="id_form-0-pubDate">Pub date:</label> <input type="text" name="form-0-pubDate" id=
</div>
*/
```

If the value of `maxNum` is greater than the number of existing objects, up to `extra` additional blank forms will be added to the formset, so long as the total number of forms does not exceed `maxNum`.

Formset validation

Validation with a formset is almost identical to a regular `Form`. There's an `isValid()` method on the formset to provide a convenient way to validate all forms in the formset:

```
var data = {
  'form-TOTAL_FORMS': '1'
, 'form-INITIAL_FORMS': '0'
, 'form-MAX_NUM_FORMS': ''
}
var formset = new ArticleFormSet({data: data})
print(formset.isValid())
// => true
```

If we provide an invalid article:

```
var data = {
  'form-TOTAL_FORMS': '2'
, 'form-INITIAL_FORMS': '0'
, 'form-MAX_NUM_FORMS': ''
, 'form-0-title': 'Test'
, 'form-0-pubDate': '1904-06-16'
, 'form-1-title': 'Test'
, 'form-1-pubDate': '' // <-- this date is missing but required
}
var formset = new ArticleFormSet({data: data})
```

```
print(formset.isValid())
// => false
print(formset.errors().map(function(e) { return e.toJSON() }))
// => [{}, {pubDate: [{message: 'This field is required.', code: 'required'}]}
```

To check how many errors there are in the formset, we can use the `totalErrorCount()` method:

```
formset.totalErrorCount()
// => 1
```

We can also check if form data differs from the initial data (i.e. the form was sent without any data):

```
var data = {
  'form-TOTAL_FORMS': '1'
, 'form-INITIAL_FORMS': '0'
, 'form-MAX_NUM_FORMS': ''
, 'form-0-title': ''
, 'form-0-pubDate': ''
}
var formset = new ArticleFormSet({data: data})
print(formset.hasChanged())
// => false
```

Understanding the ManagementForm

You may have noticed the additional data (`form-TOTAL_FORMS`, `form-INITIAL_FORMS` and `form-MAX_NUM_FORMS`) included in the formset's data above. This data is handled by the `ManagementForm`. This form defines hidden fields which are used to submit information about the number of forms in the formset. It's intended for use when a `FormSet`'s inputs are being used for a regular form submission to be handled on the server-side. If you're using newforms on the server to handle formsets bound to data from an HTTP POST and you don't provide this management data, an Error will be thrown:

```
var data = {
  'form-0-title': ''
, 'form-0-pubDate': ''
}
try {
  var formset = new ArticleFormSet({data: data})
}
catch (e) {
  print(e.message)
}
// => ManagementForm data is missing or has been tampered with
```

It is used to keep track of how many form instances are being displayed. If you are adding new forms via JavaScript, you should increment the count fields in this form as well. On the other hand, if you are using JavaScript to allow deletion of existing objects, then you need to ensure the ones being removed are properly marked for deletion by including `form-#-DELETE` in the POST data. It is expected that all forms are present in the POST data regardless.

`totalFormCount()` and `initialFormCount()`

`FormSet` has a couple of methods that are closely related to the `ManagementForm`, `totalFormCount` and `initialFormCount`.

`totalFormCount` returns the total number of forms in this formset. `initialFormCount` returns the number of forms in the formset that were pre-filled, and is also used to determine how many forms are required.

Client-side FormSets

When FormSets are used on the client-side, the `ManagementForm` isn't necessary. The formset's own form management properties are used whether or not the formset is bound.

Of particular interest is the formset's `extra` property, which can be used to implement “add another” functionality – since this is a common use case, formsets have an `addAnother()` method which does this for you.

Formsets also have a `removeForm(index)` method which takes care of the internal details of removing an extra form. *This should only ever be called with the index of an extra form in the formset.* To ensure this, if you're displaying a formset which contains both initial forms for existing data, and extra forms for new data which support deletion, use both `FormSet.initialForms()` and `FormSet.extraForms()` when rendering instead of looping over `FormSet.forms()`.

If you ever have a need to use FormSets on the client side *and* perform a regular HTTP POST request to process the form, you can still render `formset.managementForm()` – its hidden fields will be kept in sync with any changes made to the formset's form management configuration.

Updating a formset's data

Similar to Forms, a FormSet has a `formset.setData()` method which can be used to update the data bound to the formset and its forms.

This will also trigger validation – updating each form's `form.errors()` and `form.cleanedData`, and returning the result of `formset.isValid()`.

Validating a formset on-demand

To force full validation of the current state of a formset and its forms' input data, call `formset.validate()`.

Custom formset validation

A formset has a `clean()` method similar to the one on a `Form` class. This is where you define your own validation that works at the formset level:

```
var ArticleFormSet = forms.FormSet.extend({
  form: ArticleForm,
  /** Checks that no two articles have the same title. */
  clean: function() {
    if (this.totalErrorCount() !== 0) {
      // Don't bother validating the formset unless each form is valid on its own
      return
    }
    var titles = {}
    this.forms().forEach(function(form) {
      var title = form.cleanedData.title
      if (title in titles) {
        throw forms.ValidationError('Articles in a set must have distinct titles.')
      }
      titles[title] = true
    })
  }
})

var data = {
```

```

    'form-TOTAL_FORMS': '2'
  , 'form-INITIAL_FORMS': '0'
  , 'form-MAX_NUM_FORMS': ''
  , 'form-0-title': 'Test'
  , 'form-0-pubDate': '1904-06-16'
  , 'form-1-title': 'Test'
  , 'form-1-pubDate': '1912-06-23'
}
var formset = new ArticleFormSet({data: data})
print(formset.isValid())
// => false
print(formset.errors().map(function(e) { return e.toJSON() }))
// => [{}, {}]
print(formset.nonFormErrors().messages())
// => ['Articles in a set must have distinct titles.']

```

Using more than one formset in a <form>

Just like Forms, FormSets can be given a prefix to prefix form field names to allow more than one formset to be used in the same <form> without their input name attributes clashing.

For example, if we had a Book form which also had a “title” field - this is how we could avoid field names for Article and Book forms clashing:

```

var ArticleFormSet = forms.FormSet.extend({form: Article})
var BookFormSet = forms.FormSet.extend({form: Book})

var PublicationManager = React.createClass({
  getInitialState: function() {
    return {
      articleFormset: new ArticleFormSet({prefix: 'articles'})
    , bookFormset: new BookFormSet({prefix: 'books'})
    }
  },

  // ...rendering implemented as normal...

  onSubmit: function(e) {
    e.preventDefault()
    var articlesValid = this.state.articleFormset.validate()
    var booksValid = this.state.bookFormset.validate()
    if (articlesValid && booksValid) {
      // Do something with cleanedData() on the formsets
    }
  }
})

```

For server-side usage, it’s important to point out that you need to pass prefix every time you’re creating a new formset instance – on both POST and non-POST cases – so expected input names match up when submitted data is being processed.

3.1.11 Locales

New in version 0.7.

Newforms comes with two pre-configured locales: en and en_GB.

The default locale is `en`, which (for backwards-compatibility) expects any forward slash delimited date input to be in month/day/year format and will, by default, format dates as year-month-day for display in inputs.

The `en_GB` locale is provided as a quick way to switch to day/month/year date input if that's what your application needs.

Adding a new locale

To add a new locale, use `forms.addLocale()`, providing a language code and an object specifying localisation data. The following properties are expected in the locale object:

Property	Value
<code>b</code>	List of abbreviated month names
<code>B</code>	List of full month names
<code>DATE_INPUT_FORMATS</code>	Accepted date input format strings
<code>DATETIME_INPUT_FORMATS</code>	Accepted date/time input format strings
<code>TIME_INPUT_FORMATS</code>	Accepted time input format strings

For each of the `*_INPUT_FORMATS`, [ISO 8601](#) standard formats will be automatically be added if they're not already present.

For example, to add a French locale:

```
forms.addLocale('fr', {
    b: 'janv._févr._mars_avr._mai_juin_juil._août_sept._oct._nov._déc.'.split('_')
, B: 'janvier_février_mars_avril_mai_juin_juillet_août_septembre_octobre_novembre_décembre'.split('_')
, DATE_INPUT_FORMATS: [
    '%d/%m/%Y', '%d/%m/%y'
, '%d %b %Y', '%d %b %y'
, '%d %B %Y', '%d %B %y'
]
, DATETIME_INPUT_FORMATS: [
    '%d/%m/%Y %H:%M:%S'
, '%d/%m/%Y %H:%M'
, '%d/%m/%Y'
]
})
```

Setting the default locale

To set the default locale, use `forms.setDefaultLocale()`:

```
forms.setDefaultLocale('fr')
```

Fields and Widgets which deal with dates and times and haven't been explicitly configured with input/output [format strings](#) will pick up their input and output formats from the default locale the first time they need them, caching them for future use.

As such, if you want to switch locales on the fly, any form instances created prior to calling `setDefaultLocale()` should be re-initialised.

3.2 API Reference

3.2.1 Forms API

Form

`Form.extend(prototypeProps[, constructorProps])`

This is the entry point for defining your own forms.

Creates a new constructor which inherits from Form.

Arguments

- **prototypeProps** (*Object*) – form Fields and other prototype properties for the new form, such as a custom constructor and validation methods.

See *DeclarativeFieldsMeta()* for details of how Fields passed as *prototypeProps* are handled.

- **constructorProps** (*Object*) – properties to be set directly on the new constructor function.

`class Form([kwargs])`

A collection of Fields that knows how to validate and display itself.

Arguments

- **kwargs** (*Object*) – form options, which are as follows:
- **kwargs.data** (*Object*) – input form data, where property names are field names. A form with data is considered to be “bound” and ready for use validating and coercing the given data.
- **kwargs.files** (*Object*) – input file data.
- **kwargs.errors** (*ErrorObject*) – initial errors to be displayed.

Note: Passing initial errors will prevent validation from firing if a form has input `data` and `isValid()` or `errors()` are called during rendering.

This argument is intended for redisplaying a form with the user’s input and errors received from another source, such as an API call.

This is more typical of server-side usage.

- **kwargs.validation** – Configures form-wide interactive validation when the user makes changes to form inputs in the browser. This can be a String, or an Object which configures default validation for form inputs.

If `‘manual’`, interactive validation will not be performed – you are responsible for hooking up validation and using methods such as `setData()` and `isValid()` to perform all validation. This is the default setting.

If an Object is given, it should have the following properties:

on The name of the default event to use to trigger validation. For example, if `‘blur’`, text input validation will be performed when the input loses focus after editing. Multiple, space-separated event names can be given.

onChangeDelay A delay, in milliseconds, to be used to debounce performing of `onChange` validation.

If `'auto'`, validation behaviour will be the equivalent of having passed:

..code-block: javascript

```
validation: {on: 'blur change', onChangeDelay: 369}
```

If any String but `'manual'` or `'auto'` is given, it will be used as if it were passed as the `on` property of an Object.

For example, passing `{validation: 'change'}` will cause form inputs to trigger validation as soon as the user makes any change.

New in version 0.6.

- **kwargs.controlled** (*Boolean*) – Configures whether or not the form will render controlled components - when using controlled components, you can update the values displayed in the form after its initial render using `form.setData()` or `form.updateData()`

New in version 0.6.

- **kwargs.onChange** (*Function*) – If interactive validation is configured for a Form or any of its Fields, this callback function **must** be provided, or an Error will be thrown.

It will be called any time the form's input data or validation state changes as the result of user input.

Typically, this function should at least force React to update the component in which the Form is being rendered, to display the latest validation state to the user from the last change they made to the form.

New in version 0.9: Replaces `kwargs.onStateChange`

- **kwargs.autoId** (*String*) – a template for use when automatically generating `id` attributes for fields, which should contain a `{name}` placeholder for the field name – defaults to `id_{name}`.
- **kwargs.prefix** (*String*) – a prefix to be applied to the name of each field in this instance of the form - using a prefix allows you to easily work with multiple instances of the same Form object in the same HTML `<form>`, or to safely mix Form objects which have fields with the same names.
- **kwargs.initial** (*Object*) – initial form data, where property names are field names – if a field's value is not specified in `data`, these values will be used when initially rendering field widgets.
- **kwargs.errorConstructor** (*Function*) – the constructor function to be used when creating error details. Defaults to `ErrorList()`.
- **kwargs.labelSuffix** (*String*) – a suffix to be used when generating labels in one of the convenience methods which renders the entire Form – defaults to `':'`.
- **kwargs.emptyPermitted** (*Boolean*) – if `true`, the form is allowed to be empty – defaults to `false`.

Prototype Properties

Form#prefixFormat

This string defines the format used to generate name attributes for fields when a form instance is given a `prefix`. It must contain `{prefix}` and `{name}` placeholders.

The default format is `'{prefix}-{name}'`.

Type String

Instance Properties

Form options documented in `kwargs` above are all set as instance properties.

The following instance properties are also available:

`form.fields`

Form fields for this instance of the form.

Since a particular instance might want to alter its fields based on data passed to its constructor, fields given as part of the form definition are deep-copied into `fields` every time a new instance is created.

Instances should only ever modify `fields`.

Note: `fields` does not exist until the `Form` constructor has been called on the form instance that's being constructed.

This is important to note when you intend to dynamically modify `fields` when extending a form – you must call the constructor of the form which has been extended before attempting to modify `fields`.

Type Object with field names as property names and `Field` instances as properties.

`form.isInitialRender`

Determines if this form has been given input data which can be validated.

`true` if the form has data or files set.

`form.cleanedData`

After a form has been validated, it will have a `cleanedData` property. If your data does *not* validate, `cleanedData` will contain only the valid fields.

Type Object with field names as property names and valid, cleaned values coerced to the appropriate JavaScript type as properties.

Prototype Functions

Validation: Methods for validating and getting information about the results of validation:

Form#validate (`[form[, callback(err, isValid, cleanedData)]]`)

Forces the form to revalidate from scratch. If a `<form>` is given, data from it will be set on this form first. Otherwise, validation will be done with this form's current input data.

Arguments

- **form** – a `<form>` DOM node – if React's representation of the `<form>` is given, its `getDOMNode()` function will be called to get the real DOM node.
- **Boolean, Object** **callback** (`function(Error)`) – Callback for asynchronous validation.

This argument is required if the form uses asynchronous validation - an `Error` will be thrown if it's not given in this case.

The callback should be an `errback` with the signature `(err, isValid, cleanedData)`.

Returns `true` if the form only has synchronous validation and is valid.

New in version 0.6.

Changed in version 0.10: Added callback argument for async validation.

Form#fullClean()

Validates and cleans `form.data` and populates errors and `cleanedData`.

You shouldn't need to call this function directly in general use, as it's called for you when necessary by `Form#isValid()` and `Form#errors()`.

Form#partialClean(*fieldNames*)

Validates and cleans `form.data` for the given field names and triggers cross-form cleaning in case any `form.cleanedData` it uses has changed.

Arguments

- **fieldNames** (*Array*) – a list of unprefixed field names.

Form#clean([*callback(err, validationError)*])

Hook for doing any extra form-wide cleaning after each `Field#clean()` has been called. Any `ValidationError()` thrown by this method will not be associated with a particular field; it will have a special-case association with the field named `'__all__'`.

Arguments

- **String|ValidationError callback** (*function(Error;)*) – Optional callback for asynchronous validation.

Changed in version 0.10: This method can now be defined with a `callback` parameter if it needs to perform async validation. The form will provide a callback function and wait for it to be called before finishing validation.

Data mutability: Methods for programmatically changing the form's data.

Form#reset([*initialData*])

Resets the form to its initial render state, optionally giving it new initial data.

Arguments

- **initialData** (*Object*) – new initial data for the form.

New in version 0.6.

Form#setData(*data*[, *kwargs*])

Replaces the form's `form.data` with the given data (and flips `form.isInitialRender` to false, if necessary) and triggers form cleaning and validation, returning the result of `form.isValid()`.

Arguments

- **data** (*Object*) – new input data for the form
- **kwargs** (*Object*) – data updating options, which are as follows:
- **kwargs.prefixed** (*Boolean*) – pass `true` when updating data in a prefixed form and the field names in `data` are already prefixed – defaults to `false`

New in version 0.6.

Returns `true` if the form has no errors after validating the updated data, `false` otherwise.

New in version 0.5.

Form#setFormData(*formData*)

Replaces with form's input data with data extracted from a `<form>` (i.e. with `formData()`).

When using multiple forms with prefixes, form data will always be prefixed - using this method when working with manually extracted form data should ensure there are no surprises if moving from non-prefixed forms to prefixed forms.

Arguments

- **formData** (*Object*) –

new input data for the form, which has been extracted from a `<form>`

New in version 0.6.

Form#updateData (*data* [, *kwargs*])

Updates the form's `form.data` (and flips `form.isInitialRender` to `false`, if necessary).

By default, triggers validation of fields which had their input data updated, as well as form-wide cleaning.

Arguments

- **data** (*Object*) – partial input data for the form, field name -> input data.
If your form has a *prefix*, field names in the given data object must also be prefixed.
- **kwargs** (*Object*) – data updating options, which are as follows:
- **kwargs.prefixed** (*Boolean*) – pass `true` when updating data in a prefixed form and the field names in *data* are already prefixed – defaults to `false`

The following options are intended for use with controlled forms, when you're only updating data in order to change what's displayed in the controlled components:

Arguments

- **kwargs.validate** (*Boolean*) – pass `false` if you want to skip validating the updated fields – defaults to `true`. This can be ignored if you're passing known-good data.
- **kwargs.clearValidation** (*Boolean*) – pass `false` if you're skipping validation and you also want to skip clearing of the results of any previous validation on the fields being updated, such as error messages and `cleanedData` – defaults to `true`

New in version 0.6.

BoundFields: Methods which create `BoundField` helpers for rendering the form's fields.

Form#boundFields ([*test*])

Creates a `BoundField()` for each field in the form, in the order in which the fields were created.

Arguments

- **test** (*Function(field,name)*) – If provided, this function will be called with `field` and `name` arguments - `BoundFields` will only be generated for fields for which `true` is returned.

Form#boundFieldsObj ([*test*])

A version of `Form#boundFields()` which returns an `Object` with field names as property names and `BoundFields` as properties.

Form#boundField (*name*)

Creates a `BoundField()` for the field with the given name.

Arguments

- **name** (*String*) – the name of a field in the form.

Form#hiddenFields ()

Returns a list of `BoundField()` objects that correspond to hidden fields. Useful for manual form layout.

Form#visibleFields ()

Returns a list of `BoundField()` objects that do not correspond to hidden fields. The opposite of the `Form#hiddenFields()` function.

Error: Methods for working with the form's validation errors.

Form#addError (*field*, *error*)

This function allows adding errors to specific fields from within the `form.clean()` method, or from outside the form altogether.

Arguments

- **field** (*String*) – the name of the field to which the error(s) should be added. If its value is `null` the error will be treated as a non-field error as returned by `form.nonFieldErrors()`.
- **error** (*String|Array|ValidationError|Object*) – the error argument can be a single error, a list of errors, or an object that maps field names to lists of errors. A single error can be given as a `String` or an instance of a `ValidationError()`.

Multiple errors can be given as an `Array`, an `Object` which maps field names to validation errors, or a `ValidationError` created with an `Array` or `Object`.

If the `error` argument is an `Object`, the `field` argument *must* be `null` – errors will be added to the fields that correspond to the properties of the object.

Note: Using `form.addError()` automatically removes the relevant field from `form.cleanedData`.

New in version 0.5.

Changed in version 0.10: `addError()` will no longer add a duplicated error message for the same field. This can happen if event-based validation which runs repeatedly adds errors to a field other than that which triggered the validation, such as in a custom `clean()` method.

Form#errors ()

Getter for validation errors which first cleans the form if there are no errors defined yet.

Returns validation errors for the form, as an `ErrorObject()`

Form#nonFieldErrors ()

Returns errors that aren't associated with a particular field - i.e., errors generated by `Form#clean()`, or by calling `Form#addError()` and passing `null` instead of a field name. Will be an empty error list object if there are none.

Form#setErrors (*errors*)

This method's intended use is replacing a Form's errors with those received from another source, such as an API call which performs additional validation.

Arguments

- **errors** (`ErrorObject`) –

Changes: methods for working with changed data.

Form#changedData ()

Returns a list of the names of fields which have differences between their initial and currently bound values.

Form#hasChanged ()

Returns `true` if data differs from initial, `false` otherwise.

Status: methods for determining the form's status:

Form#isAsync ()

Returns `true` if the form's prototype defines any custom cleaning methods which have an arity of 1 (which is assumed to mean they have defined an async callback parameter).

New in version 0.10.

Form#isComplete()

Determines whether or not the form has valid input data for all required fields. This can be used to indicate to the user that a form which is being validated as they fill it in is ready for submission.

A form which has any errors or is pending async validation will not be considered complete.

The distinction between `isComplete()` and `Form#isValid()` is that a form which has had, for example, a single field filled in and validated is valid according to the partial validation which has been performed so far (i.e. it doesn't have any error messages) but isn't yet complete.

New in version 0.6.

Changed in version 0.10: A form which `isPending()` will not be considered complete.

Form#isMultipart()

Determines if the form needs to be multipart-encoded in other words, if it has a `FileInput()`.

Returns `true` if the form needs to be multipart-encoded.

Form#isPending()

Returns `true` if true if the form is waiting for async validation to complete.

Form#isValid()

Determines whether or not the form has errors, triggering cleaning of the form first if necessary.

When user input is being incrementally validated as it's given, this function gives you the current state of validation (i.e. whether or not there are any errors). It will not reflect the validity of the whole form until a method which performs whole-form validation (`Form#validate()` or `setData()`) has been called.

Returns `true` if the form is has input data and has no errors, `false` otherwise. If errors are being ignored, returns `false`.

Form#nonFieldPending()

Returns `true` if the form is waiting for async validation of its `clean(callback)` method to complete.

New in version 0.10.

Form#notEmpty()

Determines if a form which is an extra form in a `FormSet` has changed from its initial values. Extra forms are allowed to be empty, so required fields in them do not become truly required until the form has been modified.

Returns `true` if a form has `emptyPermitted` and has changed from its initial values.

New in version 0.9.

Prefixes: Methods for working with form prefixes.

Form#addPrefix(fieldName)

Returns the given field name with a prefix added, if this Form has a prefix.

Form#addInitialPrefix(fieldName)

Adds an initial prefix for checking dynamic initial values.

Form#removePrefix(fieldName)

Returns the given field name with a prefix-size chunk chopped off the start if this form has a prefix set and the field name starts with it.

Utilities

DeclarativeFieldsMeta (*prototypeProps*)

This mixin function is responsible for setting up form fields when a new Form constructor is being created.

It pops any Fields it finds off the form's prototype properties object, determines if any forms are also being mixed-in via a `__mixins__` property and handles inheritance of Fields from any form which is being directly extended, such that fields will be given the following order of precedence should there be a naming conflict with any of these three sources:

- 1.Fields specified in `prototypeProps`
- 2.Fields from a mixed-in form
- 3.Fields from the Form being inherited from

If multiple forms are provided via `__mixins__`, they will be processed from left to right in order of precedence for mixing in fields and prototype properties.

Forms can prevent fields from being inherited or mixed in by adding a same-named property to their prototype, which isn't a Field. It's suggested that you use `null` as the value when shadowing to make this intent more explicit.

isFormAsync (*Form*)

Arguments

- **Form** (*Form*) – a Form constructor

Returns `true` if the given Form constructor's prototype defines any custom cleaning methods which have an arity of 1 (which is assumed to mean they have defined an async callback parameter).

3.2.2 BoundField API

BoundField

class BoundField (*form, field, name*)

A helper for rendering a field.

This is the primary means of generating components such as labels and input fields in the default form rendering methods.

Its attributes and methods will be of particular use when implementing custom form layout and rendering.

Arguments

- **form** (*Form*) – the form instance the field is a part of.
- **field** (*Field*) – the field to be rendered.
- **name** (*String*) – the name associated with the field in the form.

Instance Attributes

`boundField.form`

`boundField.field`

`boundField.name`

`boundField.htmlName`

A version of the field's name including any prefix the form has been configured with.

Assuming your forms are configured with prefixes when needed, this should be a unique identifier for any particular field (e.g. if you need something to pass as a `key` prop to a React component).

Type String

`boundField.label`

The label the field is configured with, or a label automatically generated from the field's name.

Type String

`boundField.helpText`

Help text the field is configured with, otherwise an empty string.

Type String

Prototype Functions

Status: methods for determining the field's status.

BoundField#status()

Returns a string representing the field's current status.

Statuses are determined by checking the following conditions in order:

- 'pending' – the field has a pending async validation.
- 'error' – the field has a validation error.
- 'valid' – the field has a value in `form.cleanedData`.
- 'default' – the field meets none of the above criteria, i.e. it hasn't been interacted with yet, or the whole form hasn't been validated yet.

BoundField#isCleaned()

Returns `true` if the field has some data in its form's `cleanedData`.

BoundField#isEmpty()

Returns `true` if the value which will be displayed in the field's widget is empty.

BoundField#isPending()

Returns `true` if the field has a pending asynchronous validation.

BoundField#isHidden()

Returns `true` if the field is configured with a hidden widget.

Field data: methods for accessing data related to the field.

BoundField#autoId()

Calculates and returns the `id` attribute for this `BoundField` if the associated form has an `autoId` set, or set to `true`. Returns an empty string otherwise.

BoundField#data()

Returns Raw input data for the field or `null` if it wasn't given.

BoundField#errors()

Returns validation errors for the field - if there were none, an empty error list object will be returned.

Type `ErrorList()` (by default, but configurable via `Form()` `kwargs.errorConstructor`)

BoundField#errorMessage()

Convenience method for getting the first error message for the field, as a single error message is the most common error scenario for a field.

Returns the first validation error message for the field - if there were none, returns `undefined`.

BoundField#errorMessages()

Returns all validation error messages for the field - if there were none, returns an empty list.

BoundField#idForLabel()

Wrapper around the field widget's `Widget#idForLabel()`. Useful, for example, for focusing on this field regardless of whether it has a single widget or a `MutiWidget()`.

BoundField#initialValue()

Returns the initial value for the field, will be null if none was configured on the field or given to the form.

BoundField#value()

Returns the value to be displayed in the field's widget.

Rendering:: methods for, and related to, rendering a widget for the field.

BoundField#asWidget([kwargs])

Renders a widget for the field.

Arguments

- **kwargs** (*Object*) – widget options, which are as follows:
- **kwargs.widget** (*Widget*) – an override for the widget used to render the field - if not provided, the field's configured widget will be used.
- **kwargs.attrs** (*Object*) – additional HTML attributes to be added to the field's widget.

BoundField#asHidden([kwargs])

Renders the field as a hidden field.

Arguments

- **kwargs** (*Object*) – widget options, which are as follows
- **kwargs.attrs** (*Object*) – additional HTML attributes to be added to the field's widget.

BoundField#asText([kwargs])

Renders the field as a text input.

Arguments

- **kwargs** (*Object*) – widget options, which are as follows:
- **kwargs.attrs** (*Object*) – additional HTML attributes to be added to the field's widget.

BoundField#asTextarea([kwargs])

Renders the field as a textarea.

Arguments

- **kwargs** (*Object*) – widget options, which are as follows:
- **kwargs.attrs** (*Object*) – additional HTML attributes to be added to the field's widget.

BoundField#cssClasses([extraClasses])

Returns a string of space-separated CSS classes to be applied to the field.

Arguments

- **extraClasses** (*String*) – additional CSS classes to be applied to the field

BoundField#helpTextTag (*[kwargs]*)

Renders a tag containing help text for the field.

Arguments

- **kwargs** (*Object*) – help text tag options, which are as follows:
- **kwargs.tagName** (*String*) – allows overriding the type of tag – defaults to 'span'.
- **kwargs.contents** (*String|Object*) – help text contents – if not provided, contents will be taken from the field itself.

To render raw HTML in help text, it should be specified using the [React convention for raw HTML](#), which is to provide an object with a `__html` property:

```
{__html: 'But <strong>be careful</strong>!'}
```

- **kwargs.attrs** (*Object*) – additional attributes to be added to the tag – by default it will get a `className` of 'helpText'.

BoundField#labelTag (*[kwargs]*)

Creates a `<label>` for the field if it has an `id` attribute, otherwise generates a text label.

Arguments

- **kwargs** (*Object*) – label options, which are as follows:
- **kwargs.contents** (*String*) – custom contents for the label – if not provided, label contents will be generated from the field itself.
- **kwargs.attrs** (*Object*) – additional HTML attributes to be added to the label tag.
- **kwargs.labelSuffix** (*String*) – a custom suffix for the label.

BoundField#render (*[kwargs]*)

Default rendering method - if the field has `showHiddenInitial` set, renders the default widget and a hidden version, otherwise just renders the default widget for the field.

Arguments

- **kwargs** (*Object*) – widget options as per *BoundField#asWidget()*.

BoundField#subWidgets ()

Returns a list of *SubWidget()* objects that comprise all widgets in this *BoundField*. This really is only useful for *RadioSelect()* and *CheckboxSelectMultiple()* widgets, so that you can iterate over individual inputs when rendering.

3.2.3 Fields API

Field

class Field (*[kwargs]*)

An object that is responsible for doing validation and normalisation, or “cleaning” – for example: an *EmailField()* makes sure its data is a valid e-mail address – and makes sure that acceptable “blank” values all have the same representation.

Arguments

- **kwargs** (*Object*) – field options, which are as follows:
- **kwargs.required** (*Boolean*) – determines if the field is required – defaults to `true`.
- **kwargs.widget** (*Widget*) – overrides the widget used to render the field – if not provided, the field’s default will be used.
- **kwargs.label** (*String*) – the label to be displayed for the field - if not provided, will be generated from the field’s name.
- **kwargs.initial** – an initial value for the field to be used if none is specified by the field’s form.
- **kwargs.helpText** (*String*) – help text for the field.
- **kwargs.errorMessages** (*Object*) – custom error messages for the field, by error code.
- **kwargs.showHiddenInitial** (*Boolean*) – specifies if it is necessary to render a hidden widget with initial value after the widget.
- **kwargs.validators** (*Array.<Function>*) – list of additional validators to use - a validator is a function which takes a single value and throws a `ValidationError` if it’s invalid.
- **kwargs.widgetAttrs** (*Object*) – additional attributes for the field’s rendered widget.

New in version 0.11.

- **kwargs.cssClass** (*String*) – space-separated CSS classes to be applied to the field’s container when default rendering functions are used.
- **kwargs.custom** – this argument is provided to pass any custom metadata you require on the field, e.g. extra per-field options for a custom layout you’ve implemented. Newforms will set anything you pass for this argument in a `custom` instance property on the field.

New in version 0.5.

- **kwargs.validation** – Configures validation when the user interacts with this field’s widget in the browser.

This can be used to configure validation for only specific fields, or to override any form-wide validation that’s been configured.

Takes the same arguments as *Form’s validation configuration*

If validation configuration is given, the Form containing the Field **must** be configured with an *onChange callback*, or an Error will be thrown.

New in version 0.6.

- **controlled** (*Boolean*) – Configures whether or not the field will render a controlled component

This can be used to configure creation of controlled components for only specific fields, or to override any form-wide `controlled` that’s been configured.

New in version 0.6.

Prototype Functions

Field#prepareValue (*value*)

Hook for any pre-preparation required before a value can be used.

Field#toJavaScript (*value*)

Hook for coercing a value to an appropriate JavaScript object.

Field#isEmptyValue (*value*)

Checks for the given value being `===` one of the configured empty values for this field, plus any additional checks required due to JavaScript’s lack of a generic object equality checking mechanism.

This function will use the field’s `emptyValues` property for the `===` check – this defaults to `[null, undefined, '']` via `Field.prototype`.

If the field has an `emptyValueArray` property which is `true`, the value’s type and length will be checked to see if it’s an empty Array – this defaults to `true` via `Field.prototype`.

Field#validate (*value*)

Hook for validating a value.

Field#clean (*value*)

Validates the given value and returns its “cleaned” value as an appropriate JavaScript object.

Raises `ValidationError()` for any errors.

class CharField (*[kwargs]*)

Validates that its input is a valid string.

Arguments

- **kwargs** (*Object*) – field options additional to those specified in `Field()`:
- **kwargs.maxLength** (*Number*) – a maximum valid length for the input string.
- **kwargs.minLength** (*Number*) – a minimum valid length for the input string.

Numeric fields**class IntegerField** (*[kwargs]*)

Validates that its input is a valid integer.

Arguments

- **kwargs** (*Object*) – field options additional to those specified in `Field()`:
- **kwargs.maxValue** (*Number*) – a maximum valid value for the input.
- **kwargs.minValue** (*Number*) – a minimum valid value for the input.

class FloatField (*[kwargs]*)

Validates that its input is a valid float.

Arguments

- **kwargs** (*Object*) – field options additional to those specified in `Field()`:
- **kwargs.maxValue** (*Number*) – a maximum valid value for the input.
- **kwargs.minValue** (*Number*) – a minimum valid value for the input.

class DecimalField (*[kwargs]*)

Validates that its input is a decimal number.

Arguments

- **kwargs** (*Object*) – field options additional to those specified in `Field()`:
- **kwargs.maxValue** (*Number*) – a maximum value for the input.
- **kwargs.minValue** (*Number*) – a minimum value for the input.
- **kwargs.maxDigits** (*Number*) – the maximum number of digits the input may contain.

- **kwargs.decimalPlaces** (*Number*) – the maximum number of decimal places the input may contain.

Date/Time fields

class DateField (*[kwargs]*)

Validates that its input is a date.

Normalises to a `Date` with its time fields set to zero.

Arguments

- **kwargs** (*Object*) – field options additional to those specified in `Field()`:
- **kwargs.inputFormats** (*Array.<String>*) – a list of `time.strptime()` format strings which are considered valid.

class TimeField (*[kwargs]*)

Validates that its input is a time.

Normalises to a `Date` with its date fields set to 1900-01-01.

Arguments

- **kwargs** (*Object*) – field options additional to those specified in `Field()`:
- **kwargs.inputFormats** (*Array.<String>*) – a list of `time.strptime()` format strings which are considered valid.

class DateTimeField (*[kwargs]*)

Validates that its input is a date/time.

Normalises to a `Date`.

Arguments

- **kwargs** (*Object*) – field options additional to those specified in `Field()`:
- **kwargs.inputFormats** (*Array.<String>*) – a list of `time.strptime()` format strings which are considered valid.

Format fields

class RegexField (*regex[, kwargs]*)

Validates that its input matches a given regular expression.

Arguments

- **regex** (*RegExp or String*) – a regular expression to validate input against. If a string is given, it will be compiled to a `RegExp`.
- **kwargs** (*Object*) – field options, as in `CharField()`

class EmailField (*[kwargs]*)

Validates that its input appears to be a valid e-mail address.

Arguments

- **kwargs** (*Object*) – field options, as in `CharField()`

class `IPAddressField`(`[kwargs]`)
Validates that its input is a valid IPv4 address.

Deprecated since version 0.5: use `GenericIPAddressField()` instead.

class `GenericIPAddressField`(`[kwargs]`)
Validates that its input is a valid IPv4 or IPv6 address.

Arguments

- **`kwargs`** (*Object*) – field options additional to those specified in `CharField()`
- **`kwargs.protocol`** (*String*) – determines which protocols are accepted as input. One of:
 - `'both'`
 - `'ipv4'`
 - `'ipv6'`
 Defaults to `'both'`.
- **`kwargs.unpackIPv4`** (*Boolean*) – Determines if an IPv4 address that was mapped in a compressed IPv6 address will be unpacked. Defaults to `false` and can only be set to `true` if `kwargs.protocol` is `'both'`.

class `SlugField`(`[kwargs]`)
Validates that its input is a valid slug - i.e. that it contains only letters, numbers, underscores, and hyphens.

Arguments

- **`kwargs`** (*Object*) – field options, as in `CharField()`

File fields

class `FileField`(`[kwargs]`)
Validates that its input is a valid file – the behaviour of this field varies depending on the environment newforms is running in:

On the client

If the browser supports the [File API](#), `form.files` will be populated with a `File` object and validation will be performed on its name and size. The `File` object will be available via `form.cleanedData` when valid.

Otherwise, this field can only validate that a file has been selected at all, if the field is `required`.

On the server

Validates uploaded file data from `form.files`.

The contents of `form.files` are expected to have a `name` property corresponding to the uploaded file's name and a `size` property corresponding to its size.

You will need write a wrapper to provide this information depending on how you're handling file uploads.

Arguments

- **`kwargs`** (*Object*) – field options additional to those specified in `Field()`
- **`kwargs.maxLength`** (*Number*) – maximum length of the uploaded file name.
- **`kwargs.allowEmptyFile`** (*Boolean*) – if `true`, empty files will be allowed – defaults to `false`.

class `MultipleFileField` (`[kwargs]`)

A version of `FileField` which expects to receive a list of files as input and renders to an `<input type="file" multiple>` by default.

Arguments

- **kwargs** (*Object*) – field options, as in `FileField()`

class `ImageField` (`[kwargs]`)

Validates that its input is a valid uploaded image – the behaviour of this field varies depending on the environment newforms is running in:

On the client

Validates that a file has been selected if the field is `required`.

On the server

Note: As of newform 0.5, server-side image validation has not been implemented yet – `ImageField` performs the same validation as `FileField`.

Adds an `accept="image/*"` attribute to its `<input type="file">` widget.

class `URLField` (`[kwargs]`)

Validates that its input appears to be a valid URL.

Arguments

- **kwargs** (*Object*) – field options, as in `CharField()`

Boolean fields

class `BooleanField` (`[kwargs]`)

Normalises its input to a boolean primitive.

Arguments

- **kwargs** (*Object*) – field options, as in `Field()`

class `NullBooleanField` (`[kwargs]`)

A field whose valid values are `null`, `true` and `false`.

Invalid values are cleaned to `null`.

Arguments

- **kwargs** (*Object*) – field options, as in `Field()`

Choice fields

class `ChoiceField` (`[kwargs]`)

Validates that its input is one of a valid list of choices.

Arguments

- **kwargs** (*Object*) – field options additional to those specified in `Field()`:
- **kwargs.choices** (*Array*) – a list of choices - each choice should be specified as a list containing two items; the first item is a value which should be validated against, the second item is a display value for that choice, for example:


```
{choices: [[1, 'One'], [2, 'Two']]}
```

Defaults to [].

Prototype Functions

ChoiceField#choices()

Returns the current list of choices.

ChoiceField#setChoices(choices)

Updates the list of choices on this field and on its configured widget.

class TypedChoiceField([kwargs])

A ChoiceField which returns a value coerced by some provided function.

Arguments

- **kwargs** (Object) – field options additional to those specified in *ChoiceField()*:
- **kwargs.coerce** (Function(String)) – a function which takes the string value output from ChoiceField's clean method and coerces it to another type – defaults to a function which returns the given value unaltered.
- **kwargs.emptyValue** – the value which should be returned if the selected value can be validly empty – defaults to ''.

class MultipleChoiceField([kwargs])

Validates that its input is one or more of a valid list of choices.

class TypedMultipleChoiceField([kwargs])

A MultipleChoiceField which returns values coerced by some provided function.

Arguments

- **kwargs** (Object) – field options additional to those specified in MultipleChoiceField.
- **kwargs.coerce** – (Function)
function which takes the String values output by MultipleChoiceField's toJavaScript method and coerces it to another type – defaults to a function which returns the given value unaltered.
- **kwargs.emptyValue** – (Object)
the value which should be returned if the selected value can be validly empty – defaults to ''.

class FilePathField([kwargs])

Note: As of newform 0.5, server-side logic for `FilePathField` hasn't been implemented yet.

As such, this field isn't much use yet and the API documentation below is speculative.

Allows choosing from files inside a certain directory.

Arguments

- **path** (String) – The absolute path to the directory whose contents you want listed - this directory must exist.
- **kwargs** (Object) – field options additional to those supplied in *ChoiceField()*.

- **kwargs.match** (*String or RegExp*) – a regular expression pattern – if provided, only files with names matching this expression will be allowed as choices. If a string is given, it will be compiled to a `RegExp`.
- **kwargs.recursive** (*Boolean*) – if `true`, the directory will be descended into recursively and all allowed descendants will be listed as choices – defaults to `false`.
- **kwargs.allowFiles** (*Boolean*) – if `true`, files will be listed as choices. Defaults to `true`.
- **kwargs.allowFolders** (*Boolean*) – if `true`, folders will be listed as choices. Defaults to `false`.

Slightly complex fields

class ComboField (`[kwargs]`)

A Field whose `clean()` method calls multiple Field `clean()` methods.

Arguments

- **kwargs** (*Object*) – field options additional to those specified in `Field()`.
- **kwargs.fields** (*Array.<Field>*) – fields which will be used to perform cleaning, in the order they’re given.

class MultiValueField (`[kwargs]`)

A Field that aggregates the logic of multiple Fields.

Its `clean()` method takes a “decompressed” list of values, which are then cleaned into a single value according to `this.fields`. Each value in this list is cleaned by the corresponding field – the first value is cleaned by the first field, the second value is cleaned by the second field, etc. Once all fields are cleaned, the list of clean values is “compressed” into a single value.

Subclasses should not have to implement `clean()`. Instead, they must implement `compress()`, which takes a list of valid values and returns a “compressed” version of those values – a single value.

You’ll probably want to use this with `MultiWidget()`.

Arguments

- **kwargs** (*Object*) – field options
- **kwargs.fields** (*Array.<Field>*) – a list of fields to be used to clean a “decompressed” list of values.
- **kwargs.requireAllFields** (*Boolean*) – when set to `false`, allows optional sub-fields. The required attribute for each individual field will be respected, and a new ‘incomplete’ validation error will be raised when any required fields are empty. Defaults to `true`.

class SplitDateTimeField (`[kwargs]`)

A `MultiValueField` consisting of a `DateField()` and a `TimeField()`.

3.2.4 Validation API

ValidationError

`ValidationError` is part of the `validators` module, but is so commonly used when implementing custom validation that it’s exposed as part of the top-level newforms API.

class `ValidationError` (*message*_[, *kwargs*])

A validation error, containing validation messages.

Single messages (e.g. those produced by validators) may have an associated error code and error message parameters to allow customisation by fields.

Arguments

- **message** – the message argument can be a single error, a list of errors, or an object that maps field names to lists of errors.
What we define as an “error” can be either a simple string or an instance of `ValidationError` with its message attribute set, and what we define as list or object can be an actual list or object, or an instance of `ValidationError` with its `errorList` or `errorObj` property set.
- **kwargs** (*Object*) – validation error options.
- **kwargs.code** (*String*) – a code identifying the type of single message this validation error is.
- **kwargs.params** (*Object*) – parameters to be interpolated into the validation error message. where the message contains curly-bracketed {placeholders} for parameter properties.

Prototype Functions

ValidationError#messageObj ()

Returns validation messages as an object with field names as properties.

Throws an error if this validation error was not created with a field error object.

ValidationError#messages ()

Returns validation messages as a list. If the `ValidationError` was constructed with an object, its error messages will be flattened into a list.

Errors

Validation errors for a whole form are held in an `ErrorObject`, while each field’s validation errors are (by default) held in an `ErrorList`.

class `ErrorObject` ()

A collection of field errors that knows how to display itself in various formats.

Static Functions

`ErrorObject.fromJSON` (*jsonObj*_[, *errorConstructor*])

Creates a new `ErrorObject` and populates it from an object with the same structure as that produced by this object’s `toJSON` () method.

Arguments

- **jsonObj** (*Object*) –
- **errorConstructor** (*Function*) – Constructor for creating field errors - defaults to `ErrorList` ().

Prototype Functions

ErrorObject#set (*fieldName*, *errors*)

Sets a field’s errors.

Arguments

- **fieldName** (*String*) –

- **errors** ([ErrorList](#)) – validation errors for the field.

ErrorObject#get (*fieldName*)

Gets a field's errors.

Arguments

- **fieldName** (*String*) –

ErrorObject#remove (*fieldName*)

Removes errors for a field.

Arguments

- **fieldName** (*String*) –

Returns Boolean `true` if there were errors for the field.

ErrorObject#removeAll (*fieldNames*)

Removes errors for multiple fields.

Arguments

- **fieldNames** (*Array*) –

ErrorObject#hasField (*fieldName*)

Arguments

- **fieldName** (*String*) –

Returns Boolean `true` if errors have been set for the given field.

ErrorObject#length ()

Returns the number of fields errors have been set for.

ErrorObject#isPopulated ()

Returns Boolean `true` if any fields have validation errors set.

ErrorObject#render (*[kwargs]*)

Default rendering is as a ``. See below for arguments.

ErrorObject#asUl (*[kwargs]*)

Displays error details as a ``. Returns `undefined` if this object isn't populated with any errors.

Arguments

- **kwargs** (*Object*) – rendering options, which are as follows:
- **kwargs.className** (*Object*) – CSS class name(s) for the ``, defaults to `'errorlist'`.

ErrorObject#asText ()

Displays error details as text.

ErrorObject#asData ()

Creates an “unwrapped” version of the data in the `ErrorObject` - a plain `Object` with lists of `ValidationErrors` as its properties.

ErrorObject#toJSON ()

Creates a representation of all the contents of the `ErrorObject` for serialisation, to be called by `JSON.stringify()` if this object is passed to it.

ErrorObject#fromJSON (*jsonObj[, errorConstructor]*)

Populates this `ErrorObject` from an object with the same structure as that produced by this object's `toJSON()` method.

Arguments

- **jsonObj** (*Object*) –
- **errorConstructor** (*Function*) – Constructor for creating field errors - defaults to `ErrorList()`.

class ErrorList (*list*)

A list of errors which knows how to display itself in various formats.

Static Functions

`ErrorList.fromJSON` (*jsonList*)

Creates a new `ErrorList` and populates it from a list with the same structure as that produced by this object's `toJSON()` method.

Prototype Functions

ErrorList#extend (*errorList*)

Adds more errors from the given list.

ErrorList#first ()

Returns the first error message held in the list, or undefined if the list was empty.

New in version 0.9.

ErrorList#messages ()

Returns the list of error messages held in the list, converting them from `ValidationErrors` to strings first if necessary.

ErrorList#length ()

Returns the number of errors in the list.

ErrorList#isPopulated ()

Returns `true` if the list contains any errors.

ErrorList#first ()

Returns the first error message in the list.

ErrorList#messages ()

Returns all error message in the list.

ErrorObject#render (*[kwargs]*)

Default rendering is as a ``. See below for arguments.

ErrorObject#asUl (*[kwargs]*)

Displays error details as a ``. Returns undefined if this object isn't populated with any errors.

Arguments

- **kwargs** (*Object*) – rendering options, which are as follows:
- **kwargs.className** (*Object*) – CSS class name(s) for the ``, defaults to `'errorlist'`.

ErrorList#render (*[kwargs]*)

Default rendering is as a ``. See below for arguments.

ErrorList#asUl (*[kwargs]*)

Displays errors as a ``. Returns undefined if this list isn't populated with any errors.

Arguments

- **kwargs** (*Object*) – rendering options, which are as follows:

- **kwargs.className** (*Object*) – CSS class name(s) for the , defaults to 'errorlist'.

ErrorList#asText()

Displays errors as text.

ErrorList#asData()

Creates an “unwrapped” version of the data in the ErrorList - a plain Array containing ValidationErrors.

ErrorList#toJSON()

Creates a representation of all the contents of the ErrorList for serialisation, to be called by `JSON.stringify()` if this object is passed to it.

ErrorList#fromJSON (*jsonList*)

Populates this ErrorList from a list with the same structure as that produced by this object's `toJSON()` method.

Validators

Newforms depends on the `validators` module and exposes its version of it as `forms.validators`.

Constructors in the validators module are actually validation function factories – they can be called with or without `new` and will return a Function which performs the configured validation when called.

class RegexValidator (*kwargs*)

Creates a validator which validates that input matches a regular expression.

Arguments

- **kwargs** (*Object*) – validator options, which are as follows:
- **kwargs.regex** (*RegExp or String*) – the regular expression pattern to search for the provided value, or a pre-compiled RegExp. By default, matches any string (including an empty string)
- **kwargs.message** (*String*) – the error message used by `ValidationError` if validation fails. Defaults to "Enter a valid value".
- **kwargs.code** (*String*) – the error code used by `ValidationError` if validation fails. Defaults to "invalid".
- **kwargs.inverseMatch** (*Boolean*) – the match mode for regex. Defaults to `false`.

class URLValidator (*kwargs*)

Creates a validator which validates that input looks like a valid URL.

Arguments

- **kwargs** (*Object*) – validator options, which are as follows:
- **kwargs.schemes** (*Array.<String>*) – allowed URL schemes. Defaults to ['http', 'https', 'ftp', 'ftps'].

class EmailValidator (*kwargs*)

Creates a validator which validates that input looks like a valid e-mail address.

Arguments

- **kwargs** (*Object*) – validator options, which are as follows:
- **kwargs.message** (*String*) – error message to be used in any generated `ValidationError`.
- **kwargs.code** (*String*) – error code to be used in any generated `ValidationError`.

- **kwargs.whitelist** (*Array.<String>*) – a whitelist of domains which are allowed to be the only thing to the right of the @ in a valid email address – defaults to ['localhost'].

validateEmail (*value*)

Validates that input looks like a valid e-mail address – this is a preconfigured instance of an *EmailValidator()*.

validateSlug (*value*)

Validates that input consists of only letters, numbers, underscores or hyphens.

validateIPv4Address (*value*)

Validates that input looks like a valid IPv4 address.

validateIPv6Address (*value*)

Validates that input is a valid IPv6 address.

validateIPv46Address (*value*)

Validates that input is either a valid IPv4 or IPv6 address.

validateCommaSeparatedIntegerList (*value*)

Validates that input is a comma-separated list of integers.

class MaxValueValidator (*maxValue*)

Throws a *ValidationError* with a code of 'maxValue' if its input is greater than *maxValue*.

class MinValueValidator (*minValue*)

Throws a *ValidationError* with a code of 'minValue' if its input is less than *maxValue*.

class MaxLengthValidator (*maxLength*)

Throws a *ValidationError* with a code of 'maxLength' if its input's length is greater than *maxLength*.

class MinLengthValidator (*minLength*)

Throws a *ValidationError* with a code of 'minLength' if its input's length is less than *minLength*.

3.2.5 Widgets API

Widget

class Widget (*[kwargs]*)

An HTML form widget.

A widget handles the rendering of HTML, and the extraction of data from an object that corresponds to the widget.

This base widget cannot be rendered, but provides the basic attribute *widget.attrs*. You must implement the *Widget#render`()* method when extending this base widget.

Arguments

- **kwargs** (*Object*) – widget options, which are as follows:
- **kwargs.attrs** (*Object*) – HTML attributes for the rendered widget.

Instance Properties

widget.attrs

Base HTML attributes for the rendered widget.

Type *Object*

Prototype Properties

Widget#isHidden

Determines whether this corresponds to an `<input type="hidden">`.

Type Boolean

Widget#needsMultipartForm

Determines whether this widget needs a multipart-encoded form.

Type Boolean

Widget#needsInitialValue

Determines whether this widget's render logic always needs to use the initial value.

Type Boolean

Widget#isRequired

Determines whether this widget is for a required field.

Type Boolean

Prototype Functions**Widget#subWidgets** (*name*, *value*[, *kwargs*])

Yields all “subwidgets” of this widget. Used by:

- `RadioSelect()` to allow access to individual radio inputs.
- `CheckboxSelectMultiple()` to allow access to individual checkbox inputs.

Arguments are the same as for `Widget#render()`.

Widget#render (*name*, *value*[, *kwargs*])

Returns a rendered representation of this Widget as a `ReactElement` object.

The default implementation throws an `Error` – extending widgets must provide an implementation.

The *value* given is not guaranteed to be valid input, so inheriting implementations should program defensively.

Arguments

- **name** (*String*) – the name to give to the rendered widget, or to be used as the basis for other, unique names when the widget needs to render multiple inputs.
- **value** – the value to be displayed in the widget.
- **kwargs** (*Object*) – rendering options, which are:
- **kwargs.attrs** (*Object*) – additional HTML attributes for the rendered widget.
- **kwargs.controlled** (*Boolean*) – `true` if the Widget should render a controlled component.
- **kwargs.initialValue** – if the widget has `Widget#needsInitialValue` configured to `true`, its initial value will always be passed

Widget#buildAttrs (*kwargsAttrs*, *renderAttrs*)

Helper function for building an HTML attributes object using `widget.attrs` and the given arguments.

Arguments

- **kwargsAttrs** (*Object*) – any extra HTML attributes passed to the Widget's `render()` method.
- **renderAttrs** (*Object*) – any other attributes which should be included in a Widget's HTML attributes by default – provided by the `render()` method for attributes related to the type of widget being implemented.

Widget#valueFromData (*data, files, name*)

Retrieves a value for this widget from the given form data.

Returns a value for this widget, or `null` if no value was provided.

Widget#idForLabel (*id*)

Determines the HTML `id` attribute of this Widget for use by a `<label>`, given the id of the field.

This hook is necessary because some widgets have multiple HTML elements and, thus, multiple ids. In that case, this method should return an id value that corresponds to the first id in the widget's tags.

class SubWidget (*parentWidget, name, value[, kwargs]*)

Some widgets are made of multiple HTML elements – namely, `RadioSelect()`. This represents the “inner” HTML element of a widget.

Prototype Functions

SubWidget#render ()

Calls the parent widget's render function with this Subwidget's details.

MultiWidget

class MultiWidget (*widgets[, kwargs]*)

A widget that is composed of multiple widgets.

You'll probably want to use this class with `MultiValueField()`.

Arguments

- **widgets** (*Array*) – the list of widgets composing this widget.
- **kwargs** (*Object*) – widget options.

Prototype Functions

MultiWidget#render (*name, value[, kwargs]*)

Arguments

- **name** (*String*) – the name be used as the basis for unique names for the multiple inputs this widget must render.
- **value** – the value to be displayed in the widget – may be a list of values or a single value which needs to be split for display.
- **kwargs** (*Object*) – rendering options, which are:
- **kwargs.attrs** (*Object*) – additional HTML attributes.

MultiWidget#formatOutput (*renderedWidgets*)

Creates an element containing a given list of rendered widgets.

This hook allows you to format the HTML design of the widgets, if needed – by default, they are wrapped in a `<div>`.

Arguments

- **renderedWidgets** (*Array*) – a list of rendered widgets.

MultiWidget#decompress (*value*)

This method takes a single “compressed” value from the field and returns a list of “decompressed” values. The input value can be assumed valid, but not necessarily non-empty.

This method **must be implemented** when extending `MultiWidget`, and since the value may be empty, the implementation must be defensive.

The rationale behind “decompression” is that it is necessary to “split” the combined value of the form field into the values for each widget.

An example of this is how `SplitDateTimeWidget()` turns a `Date` value into a list with date and time split into two separate values.

Text input widgets

class `Input` (`[kwargs]`)

An `<input>` widget.

class `TextInput` (`[kwargs]`)

An `<input type="text">` widget

class `NumberInput` (`[kwargs]`)

An `<input type="number">` widget

New in version 0.5.

class `EmailInput` (`[kwargs]`)

An `<input type="email">` widget

New in version 0.5.

class `URLInput` (`[kwargs]`)

An `<input type="url">` widget

New in version 0.5.

class `PasswordInput` (`[kwargs]`)

An `<input type="password">` widget.

Arguments

- **kwargs** (*Object*) – widget options
- **kwargs.renderValue** (*Boolean*) – if `false` a value will not be rendered for this field – defaults to `false`.

class `HiddenInput` (`[kwargs]`)

An `<input type="hidden">` widget.

class `Textarea` (`[kwargs]`)

A `<textarea>` widget.

Default `rows` and `cols` HTML attributes will be used if not provided in `kwargs.attrs`.

Date-formatting text input widgets

class `DateInput` (`[kwargs]`)

An `<input type="text">` which, if given a `Date` object to display, formats it as an appropriate date string.

Arguments

- **kwargs** (*Object*) – widget options
- **kwargs.format** (*String*) – a `time.strftime()` format string for a date.

class `DateTimeInput` (`[kwargs]`)

An `<input type="text">` which, if given a `Date` object to display, formats it as an appropriate datetime string.

Arguments

- **kwargs** (*Object*) – widget options
- **kwargs.format** (*String*) – a `time.strftime()` format string for a datetime.

class TimeInput (`[kwargs]`)

An `<input type="text">` which, if given a Date object to display, formats it as an appropriate time string.

Arguments

- **kwargs** (*Object*) – widget options
- **kwargs.format** (*String*) – a `time.strftime()` format string for a time.

Selector and checkbox widgets

class CheckboxInput (`[kwargs]`)

An `<input type="checkbox">` widget.

Arguments

- **kwargs** (*Object*) – widget options
- **kwargs.checkTest** (*Function*) – a function which takes a value and returns `true` if the checkbox should be checked for that value.

class Select (`[kwargs]`)

An HTML `<select>` widget.

Arguments

- **kwargs** (*Object*) – widget options
- **kwargs.choices** (*Array*) – choices to be used when rendering the widget, with each choice specified as pair in `[value, text]` format – defaults to `[]`.

class NullBooleanSelect (`[kwargs]`)

A `<select>` widget intended to be used with `NullBooleanField()`.

Any `kwargs.choices` provided will be overridden with the specific choices this widget requires.

class SelectMultiple (`[kwargs]`)

An HTML `<select>` widget which allows multiple selections.

Arguments

- **kwargs** (*Object*) – widget options, as per `Select()`.

class RadioSelect (`[kwargs]`)

Renders a single select as a list of `<input type="radio">` elements.

Arguments

- **kwargs** (*Object*) – widget options
- **kwargs.renderer** (*Function*) – a custom `RadioFieldRenderer()` constructor.

Prototype Functions

RadioSelect#getRenderer (*name*, *value*`[, kwargs]`)

Returns an instance of the renderer to be used to render this widget.

RadioSelect#subWidgets (*name*, *value*`[, kwargs]`)

Returns a list of `RadioChoiceInput()` objects created by this widget's renderer.

class RadioFieldRenderer (*name, value, attrs, choices*)

An object used by [RadioSelect\(\)](#) to enable customisation of radio widgets.

Arguments

- **name** (*String*) – the field name.
- **value** (*String*) – the selected value.
- **attrs** (*Object*) – HTML attributes for the widget.
- **choices** (*Array*) – choices to be used when rendering the widget, with each choice specified as an Array in `[value, text]` format.

RadioFieldRenderer#choiceInputs ()

gets all `RadioChoiceInput` inputs created by this renderer.

RadioFieldRenderer#choiceInput (*i*)

gets the *i*-th `RadioChoiceInput` created by this renderer.

class RadioChoiceInput (*name, value, attrs, choice, index*)

An object used by [RadioFieldRenderer\(\)](#) that represents a single `<input type="radio">`.

Arguments

- **name** (*String*) – the field name.
- **value** (*String*) – the selected value.
- **attrs** (*Object*) – HTML attributes for the widget.
- **choice** (*Array*) – choice details to be used when rendering the widget, specified as an Array in `[value, text]` format.
- **index** (*Number*) – the index of the radio button this widget represents.

class CheckboxSelectMultiple (*[kwargs]*)

Multiple selections represented as a list of `<input type="checkbox">` widgets.

Arguments

- **kwargs** (*Object*) – widget options
- **kwargs.renderer** (*Function*) – a custom [CheckboxFieldRenderer\(\)](#) constructor.

Prototype Functions

CheckboxSelectMultiple#getRenderer (*name, value[, kwargs]*)

Returns an instance of the renderer to be used to render this widget.

CheckboxSelectMultiple#subWidgets (*name, value[, kwargs]*)

Returns a list of [CheckboxChoiceInput\(\)](#) objects created by this widget's renderer.

class CheckboxFieldRenderer (*name, value, attrs, choices*)

An object used by [CheckboxSelectMultiple\(\)](#) to enable customisation of checkbox widgets.

Arguments

- **name** (*String*) – the field name.
- **value** (*Array*) – a list of selected values.
- **attrs** (*Object*) – HTML attributes for the widget.

- **choices** (*Array*) – choices to be used when rendering the widget, with each choice specified as an Array in `[value, text]` format.

CheckboxFieldRenderer#choiceInputs ()
gets all `CheckboxChoiceInput` inputs created by this renderer.

CheckboxFieldRenderer#choiceInput (*i*)
gets the *i*-th `CheckboxChoiceInput` created by this renderer.

class CheckboxChoiceInput (*name, value, attrs, choice, index*)
An object used by `CheckboxFieldRenderer` () that represents a single `<input type="checkbox">`.

Arguments

- **name** (*String*) – the field name.
- **value** (*Array*) – a list of selected values.
- **attrs** (*Object*) – HTML attributes for the widget.
- **choice** (*Array*) – choice details to be used when rendering the widget, specified as an Array in `[value, text]` format.
- **index** (*Number*) – the index of the checkbox this widget represents.

File upload widgets

class FileInput (*[kwargs]*)
An `<input type="file">` widget.

class ClearableFileInput (*[kwargs]*)
A file widget which also has a checkbox to indicate that the field should be cleared.

Composite widgets

class MultipleHiddenInput (*[kwargs]*)
A widget that handles `<input type="hidden">` for fields that have a list of values.

class SplitDateTimeWidget (*[kwargs]*)
Splits Date input into two `<input type="text">` elements.

Arguments

- **kwargs** (*Object*) – widget options additional to those specified in `MultiWidget` ().
- **kwargs.dateFormat** (*String*) – a `time.strftime()` format string for a date.
- **kwargs.timeFormat** (*String*) – a `time.strftime()` format string for a time.

class SplitHiddenDateTimeWidget (*[kwargs]*)
Splits Date input into two `<input type="hidden">` elements.

3.2.6 Formsets API

FormSet

class FormSet (*[kwargs]*)
A collection of instances of the same Form.

Arguments

- **kwargs** (*Object*) – configuration options.

FormSet options

New in version 0.11: FormSet options and defaulting from the prototype replace the use of `formsetFactory`, which was removed in version 0.12.

The following options configure the FormSet itself.

Default values can be pre-configured by extending the FormSet with `FormSet.extend()` to set them as prototype props.

Alternatively the base `FormSet` can be used directly, passing all FormSet and Form arguments at the same time.

Arguments

- **form** (*Function*) – the constructor for the Form to be managed.
- **kwargs.extra** (*Number*) – the number of extra forms to be displayed – defaults to 1.
- **kwargs.canOrder** (*Boolean*) – if `true`, forms can be ordered – defaults to `false`.
- **kwargs.canDelete** (*Boolean*) – if `true`, forms can be deleted – defaults to `false`.
- **kwargs.maxNum** (*Number*) – the maximum number of forms to be displayed – defaults to `DEFAULT_MAX_NUM`.
- **kwargs.validateMax** (*Boolean*) – if `true`, validation will also check that the number of forms in the data set, minus those marked for deletion, is less than or equal to `maxNum`.
- **kwargs.minNum** (*Number*) – the minimum number of forms to be displayed – defaults to 0.
- **kwargs.validateMin** (*Boolean*) – if `true`, validation will also check that the number of forms in the data set, minus those marked for deletion, is greater than or equal to `minNum`.

Form options

The following options are used when constructing forms for the formset.

Arguments

- **kwargs.data** (*Array.<Object>*) – list of input form data for each form, where property names are field names. A formset with data is considered to be “bound” and ready for use validating and coercing the given data.
- **kwargs.files** (*Array.<Object>*) – list of input file data for each form.
- **kwargs.autoId** (*String*) – a template for use when automatically generating `id` attributes for fields, which should contain a `{name}` placeholder for the field name. Defaults to `id_{name}`.
- **kwargs.prefix** (*String*) – a prefix to be applied to the name of each field in each form instance.
- **kwargs.onChange** (*Function*) – A callback to indicate to the a React component that the formset has changed, e.g. when another form is added to the formset.

This will be passed as the *Form constructor’s onChange argument* when creating the formset’s forms.

New in version 0.9: Replaces `kwargs.onStateChange`

- **kwargs.initial** (*Array.<Object>*) – a list of initial form data objects, where property names are field names – if a field's value is not specified in `data`, these values will be used when rendering field widgets.
- **kwargs.errorConstructor** (*Function*) – the constructor function to be used when creating error details - defaults to `ErrorList()`.
- **kwargs.validation** (*Function*) – A value to be passed as the *Form constructor's validation argument* when creating the formset's forms – defaults to `null`.

New in version 0.6.

- **kwargs.managementFormCssClass** (*String*) – a CSS class to be applied when rendering `FormSet#managementForm()`, as default rendering methods place its hidden fields in an additional form row just for hidden fields, to ensure valid markup is generated.

`FormSet.extend(prototypeProps[, constructorProps])`

Creates a new constructor which inherits from `FormSet`.

Arguments

- **prototypeProps** (*Object*) – Default `FormSet` options and other prototype properties for the new formset, such as a custom `clean` method.
- **constructorProps** (*Object*) – properties to be set directly on the new constructor function.

Prototype Properties

FormSet#prefixFormat

This string defines the format used to generate a `prefix` for forms in the formset to ensure they have unique name attributes. It must contain `{prefix}` and `{index}` placeholders.

The default format is `'{prefix}-{index}'`.

Type String

Instance Properties

Formset options documented in `kwargs` above are set as instance properties.

The following instance properties are also available:

formset.isInitialRender

Determines if this formset has been given input data which can be validated, or if it will display as blank or with configured initial values the first time it's rendered.

`false` if the formset was instantiated with `kwargs.data` or `kwargs.files`, `true` otherwise.

Prototype Functions

Prototype functions for retrieving forms and information about forms which will be displayed.

FormSet#managementForm()

Creates and returns the `ManagementForm` instance for this formset.

A `ManagementForm` contains hidden fields which are used to keep track of how many form instances are displayed on the page.

Browser-specific On the browser, `ManagementForms` will only ever contain `initial` data reflecting the formset's own configuration properties.

FormSet#totalFormCount()

Determines the number of form instances this formset contains, based on either submitted management data or initial configuration, as appropriate.

Browser-specific On the browser, only the formset's own form count configuration will be consulted.

FormSet#initialFormCount()

Determines the number of initial form instances this formset contains, based on either submitted management data or initial configuration, as appropriate.

Browser-specific On the browser, only the formset's own form count configuration will be consulted.

FormSet#forms()

Returns a list of this formset's form instances.

FormSet#addAnother()

Increments `formset.extra` and adds another form to the formset.

FormSet#removeForm(index)

Decrements `formset.extra` and removes the form at the specified index from the formset.

You must ensure the UI never lets the user remove anything but extra forms.

New in version 0.9.

FormSet#initialForms()

Returns a list of all the initial forms in this formset.

FormSet#extraForms()

Returns a list of all the extra forms in this formset.

FormSet#emptyForm()

Creates an empty version of one of this formset's forms which uses a placeholder '`__prefix__`' prefix – this is intended for cloning on the client to add more forms when newforms is only being used on the server.

Prototype functions for validating and getting information about the results of validation, and for retrieving forms based on submitted data:

FormSet#validate([form[, callback(err, isValid, cleanedData)])

Forces the formset to revalidate from scratch. If a `<form>` is given, data from it will be set on the formset's forms. Otherwise, validation will be done with each form's current input data.

Arguments

- **form** – a `<form>` DOM node – if React's representation of the `<form>` is given, its `ReactDOMNode()` function will be called to get the real DOM node.
- **Boolean, Object) callback** (*function(Error;)*) – Callback for asynchronous validation.

This argument is required if the formdet or its form uses asynchronous validation - an Error will be thrown if it's not given in this case.

The callback should be an errback with the signature `(err, isValid, cleanedData)`.

Returns `true` if the formset's forms' data is valid, `false` otherwise.

New in version 0.9.

Changed in version 0.10: Added callback argument for async validation.

FormSet#setData(data)

Updates the formset's `formset.data` (and `formset.isInitialRender`, if necessary) and triggers form cleaning and validation, returning the result of `formset.isValid()`.

Arguments

- **data** (*Object*) – new input data for the formset.

Returns `true` if the formset has no errors after validating the updated data, `false` otherwise.

New in version 0.5.

FormSet#setFormData (*formData*)

Alias for `FormSet#setData()`, to keep the FormSet API consistent with the Form API.

New in version 0.6.

FormSet#cleanedData ()

Returns a list of `form.cleanedData` objects for every form in `FormSet#forms()`.

Changed in version 0.9: No longer returns cleaned data for extra forms which haven't been modified.

Changed in version 0.10: No longer includes cleaned data from incomplete extra forms.

FormSet#deletedForms ()

Returns a list of forms that have been marked for deletion.

FormSet#orderedForms ()

Returns a list of forms in the order specified by the incoming data.

Throws an Error if ordering is not allowed.

FormSet#addError (*errpr*)

Adds an error that isn't associated with a particular form.

The *error* argument can be a simple string, or an instance of `ValidationError()`.

New in version 0.9.

FormSet#nonFormErrors ()

Returns an `ErrorList()` of errors that aren't associated with a particular form – i.e., from `FormSet#clean()` or externally via `FormSet#addError()`.

Returns an empty `ErrorList()` if there are none.

FormSet#errors ()

Returns a list of form error for every form in the formset.

FormSet#totalErrorCount ()

Returns the number of errors across all forms in the formset.

FormSet#isValid ()

Returns `true` if every form in the formset is valid.

FormSet#fullClean ()

Cleans all of this.data and populates formset error objects.

FormSet#clean ()

Hook for doing any extra formset-wide cleaning after `Form.clean()` has been called on every form.

Any `ValidationError()` raised by this method will not be associated with a particular form; it will be accessible via `js:func:FormSet#nonFormErrors`

FormSet#hasChanged ()

Returns `true` if any form differs from initial.

Prototype functions for use in rendering forms.

FormSet#getDefaultPrefix ()

Returns the default base prefix for each form: `'form'`.

FormSet#addFields (*form*, *index*)

A hook for adding extra fields on to a form instance.

Arguments

- **form** (*Form*) – the form fields will be added to.
- **index** (*Number*) – the index of the given form in the formset.

FormSet#addPrefix (*index*)

Returns a formset prefix with the given form index appended.

Arguments

- **index** (*Number*) – the index of a form in the formset.

FormSet#isMultipart ()

Returns `true` if the formset needs to be multipart-encoded, i.e. it has a `FileInput()`. Otherwise, `false`.

DEFAULT_MAX_NUM

The default maximum number of forms in a formset is 1000, to protect against memory exhaustion.

3.2.7 Utilities

Newforms exposes various utilities you may want to make use of when working with forms, as well as some implementation details which you may need to make use of for customisation purposes.

getFormData (*form*)**Arguments**

- **form** – a form DOM node or `ReactElement`.

Returns an object containing the submittable value(s) held in each of the form's elements, with element names as properties.

validateAll (*form*, *formsAndFormsets*)

Extracts data from a `<form>` and validates it with a list of `Forms` and/or `FormSets`.

Arguments

- **form** – the `<form>` into which any given forms and formsets have been rendered – this can be a `React <form>` component or a real `<form>` DOM node.
- **formsAndFormsets** (*Array*) – a list of `Form` and/or `FormSet` instances to be used to validate the `<form>`'s input data.

Returns `true` if the `<form>`'s input data are valid according to all given forms and formsets

util.formatToArray (*str*, *obj* [, *options*])

Replaces `{placeholders}` in a string with same-named properties from a given `Object`, but interpolates into and returns an `Array` instead of a `String`.

By default, any resulting empty strings are stripped out of the `Array` before it is returned. To disable this, pass an options object with a `'strip'` property which is `false`.

This is useful for simple templating which needs to include `ReactElement` objects.

Arguments

- **str** (*String*) – a `String` containing placeholder names surrounded by `{ }`
- **obj** (*Object*) – an `Object` whose properties will provide replacements for placeholders

- **options** (*Object*) – an options Object which can be used to disable stripping of empty strings from the resulting Array before it is returned by passing `{strip: false}`

`util.makeChoices` (*list*, *submitValueProp*, *displayValueProp*)

Creates a list of [submitValue, displayValue] *choice pairs* from a list of objects.

If any of the property names correspond to a function in an object, the function will be called with the object as the `this` context.

Arguments

- **list** (*Array*) – a list of objects
- **submitValueProp** (*String*) – the name of the property in each object holding the value to be submitted/returned when it's a selected choice.
- **displayValueProp** (*String*) – the name of the property in each object holding the value to be displayed for selection by the user.

B

BooleanField() (class), 92
 BoundField() (class), 84
 boundField.field (boundField attribute), 84
 boundField.form (boundField attribute), 84
 boundField.helpText (boundField attribute), 85
 boundField.htmlName (boundField attribute), 85
 boundField.label (boundField attribute), 85
 boundField.name (boundField attribute), 84
 BoundField#asHidden() (built-in function), 86
 BoundField#asText() (built-in function), 86
 BoundField#asTextarea() (built-in function), 86
 BoundField#asWidget() (built-in function), 86
 BoundField#autoId() (built-in function), 85
 BoundField#cssClasses() (built-in function), 86
 BoundField#data() (built-in function), 85
 BoundField#errorMessage() (built-in function), 86
 BoundField#errorMessages() (built-in function), 86
 BoundField#errors() (built-in function), 85
 BoundField#helpTextTag() (built-in function), 87
 BoundField#idForLabel() (built-in function), 86
 BoundField#initialValue() (built-in function), 86
 BoundField#isCleaned() (built-in function), 85
 BoundField#isEmpty() (built-in function), 85
 BoundField#isHidden() (built-in function), 85
 BoundField#isPending() (built-in function), 85
 BoundField#labelTag() (built-in function), 87
 BoundField#render() (built-in function), 87
 BoundField#status() (built-in function), 85
 BoundField#subWidgets() (built-in function), 87
 BoundField#value() (built-in function), 86

C

CharField() (class), 89
 CheckboxChoiceInput() (class), 105
 CheckboxFieldRenderer() (class), 104
 CheckboxFieldRenderer#choiceInput() (built-in function), 105
 CheckboxFieldRenderer#choiceInputs() (built-in function), 105

CheckboxInput() (class), 103
 CheckboxSelectMultiple() (class), 104
 CheckboxSelectMultiple#getRenderer() (built-in function), 104
 CheckboxSelectMultiple#subWidgets() (built-in function), 104
 ChoiceField() (class), 92
 ChoiceField#choices() (built-in function), 93
 ChoiceField#setChoices() (built-in function), 93
 ClearableFileInput() (class), 105
 ComboField() (class), 94

D

DateField() (class), 90
 DateInput() (class), 102
 DateTimeField() (class), 90
 DateTimeInput() (class), 102
 DecimalField() (class), 89
 DeclarativeFieldsMeta() (built-in function), 84
 DEFAULT_MAX_NUM (global variable or constant), 110

E

EmailField() (class), 90
 EmailInput() (class), 102
 EmailValidator() (class), 98
 ErrorList() (class), 97
 ErrorList.fromJSON() (ErrorList method), 97
 ErrorList#asData() (built-in function), 98
 ErrorList#asText() (built-in function), 98
 ErrorList#asUl() (built-in function), 97
 ErrorList#extend() (built-in function), 97
 ErrorList#first() (built-in function), 97
 ErrorList#fromJSON() (built-in function), 98
 ErrorList#isPopulated() (built-in function), 97
 ErrorList#length() (built-in function), 97
 ErrorList#messages() (built-in function), 97
 ErrorList#render() (built-in function), 97
 ErrorList#toJSON() (built-in function), 98
 ErrorObject() (class), 95
 ErrorObject.fromJSON() (ErrorObject method), 95

ErrorObject#asData() (built-in function), 96
ErrorObject#asText() (built-in function), 96
ErrorObject#asUI() (built-in function), 96, 97
ErrorObject#fromJSON() (built-in function), 96
ErrorObject#get() (built-in function), 96
ErrorObject#hasField() (built-in function), 96
ErrorObject#isPopulated() (built-in function), 96
ErrorObject#length() (built-in function), 96
ErrorObject#remove() (built-in function), 96
ErrorObject#removeAll() (built-in function), 96
ErrorObject#render() (built-in function), 96, 97
ErrorObject#set() (built-in function), 95
ErrorObject#toJSON() (built-in function), 96

F

Field() (class), 87
Field#clean() (built-in function), 89
Field#isEmptyValue() (built-in function), 88
Field#prepareValue() (built-in function), 88
Field#toJavaScript() (built-in function), 88
Field#validate() (built-in function), 89
FileField() (class), 91
FileInput() (class), 105
FilePathField() (class), 93
FloatField() (class), 89
Form() (class), 77
form.cleanedData (form attribute), 79
Form.extend() (Form method), 77
form.fields (form attribute), 79
form.isInitialRender (form attribute), 79
Form#addError() (built-in function), 82
Form#addInitialPrefix() (built-in function), 83
Form#addPrefix() (built-in function), 83
Form#boundField() (built-in function), 81
Form#boundFields() (built-in function), 81
Form#boundFieldsObj() (built-in function), 81
Form#changedData() (built-in function), 82
Form#clean() (built-in function), 80
Form#errors() (built-in function), 82
Form#fullClean() (built-in function), 79
Form#hasChanged() (built-in function), 82
Form#hiddenFields() (built-in function), 81
Form#isAsync() (built-in function), 82
Form#isComplete() (built-in function), 83
Form#isMultipart() (built-in function), 83
Form#isPending() (built-in function), 83
Form#isValid() (built-in function), 83
Form#nonFieldErrors() (built-in function), 82
Form#nonFieldPending() (built-in function), 83
Form#notEmpty() (built-in function), 83
Form#partialClean() (built-in function), 80
Form#prefixFormat (None attribute), 78
Form#removePrefix() (built-in function), 83
Form#reset() (built-in function), 80

Form#setData() (built-in function), 80
Form#setErrors() (built-in function), 82
Form#setFormData() (built-in function), 80
Form#updateData() (built-in function), 81
Form#validate() (built-in function), 79
Form#visibleFields() (built-in function), 81
FormSet() (class), 105
FormSet.extend() (FormSet method), 107
formset.isInitialRender (formset attribute), 107
FormSet#addAnother() (built-in function), 108
FormSet#addError() (built-in function), 109
FormSet#addFields() (built-in function), 109
FormSet#addPrefix() (built-in function), 110
FormSet#clean() (built-in function), 109
FormSet#cleanedData() (built-in function), 109
FormSet#deletedForms() (built-in function), 109
FormSet#emptyForm() (built-in function), 108
FormSet#errors() (built-in function), 109
FormSet#extraForms() (built-in function), 108
FormSet#forms() (built-in function), 108
FormSet#fullClean() (built-in function), 109
FormSet#getDefaultPrefix() (built-in function), 109
FormSet#hasChanged() (built-in function), 109
FormSet#initialFormCount() (built-in function), 108
FormSet#initialForms() (built-in function), 108
FormSet#isMultipart() (built-in function), 110
FormSet#isValid() (built-in function), 109
FormSet#managementForm() (built-in function), 107
FormSet#nonFormErrors() (built-in function), 109
FormSet#orderedForms() (built-in function), 109
FormSet#prefixFormat (None attribute), 107
FormSet#removeForm() (built-in function), 108
FormSet#setData() (built-in function), 108
FormSet#setFormData() (built-in function), 109
FormSet#totalErrorCount() (built-in function), 109
FormSet#totalFormCount() (built-in function), 107
FormSet#validate() (built-in function), 108

G

GenericIPAddressField() (class), 91
getFormData() (built-in function), 110

H

HiddenInput() (class), 102

I

ImageField() (class), 92
Input() (class), 102
IntegerField() (class), 89
IPAddressField() (class), 90
isFormAsync() (built-in function), 84

M

MaxLengthValidator() (class), 99

MaxValueValidator() (class), 99
 MinLengthValidator() (class), 99
 MinValueValidator() (class), 99
 MultipleChoiceField() (class), 93
 MultipleFileField() (class), 92
 MultipleHiddenInput() (class), 105
 MultiValueField() (class), 94
 MultiWidget() (class), 101
 MultiWidget#decompress() (built-in function), 101
 MultiWidget#formatOutput() (built-in function), 101
 MultiWidget#render() (built-in function), 101

N

NullBooleanField() (class), 92
 NullBooleanSelect() (class), 103
 NumberInput() (class), 102

P

PasswordInput() (class), 102

R

RadioChoiceInput() (class), 104
 RadioFieldRenderer() (class), 104
 RadioFieldRenderer#choiceInput() (built-in function), 104
 RadioFieldRenderer#choiceInputs() (built-in function), 104
 RadioSelect() (class), 103
 RadioSelect#getRenderer() (built-in function), 103
 RadioSelect#subWidgets() (built-in function), 103
 RegexField() (class), 90
 RegexValidator() (class), 98
 RFC
 RFC 4291#section-2.2, 49

S

Select() (class), 103
 SelectMultiple() (class), 103
 SlugField() (class), 91
 SplitDateTimeField() (class), 94
 SplitDateTimeWidget() (class), 105
 SplitHiddenDateTimeWidget() (class), 105
 SubWidget() (class), 101
 SubWidget#render() (built-in function), 101

T

Textarea() (class), 102
 TextInput() (class), 102
 TimeField() (class), 90
 TimeInput() (class), 103
 TypedChoiceField() (class), 93
 TypedMultipleChoiceField() (class), 93

U

URLField() (class), 92
 URLInput() (class), 102
 URLValidator() (class), 98
 util.formatToArray() (util method), 110
 util.makeChoices() (util method), 111

V

validateAll() (built-in function), 110
 validateCommaSeparatedIntegerList() (built-in function), 99
 validateEmail() (built-in function), 99
 validateIPv46Address() (built-in function), 99
 validateIPv4Address() (built-in function), 99
 validateIPv6Address() (built-in function), 99
 validateSlug() (built-in function), 99
 ValidationError() (class), 94
 ValidationError#messageObj() (built-in function), 95
 ValidationError#messages() (built-in function), 95

W

Widget() (class), 99
 widget.attrs (widget attribute), 99
 Widget#buildAttrs() (built-in function), 100
 Widget#idForLabel() (built-in function), 101
 Widget#isHidden (None attribute), 99
 Widget#isRequired (None attribute), 100
 Widget#needsInitialValue (None attribute), 100
 Widget#needsMultipartForm (None attribute), 100
 Widget#render() (built-in function), 100
 Widget#subWidgets() (built-in function), 100
 Widget#valueFromData() (built-in function), 100