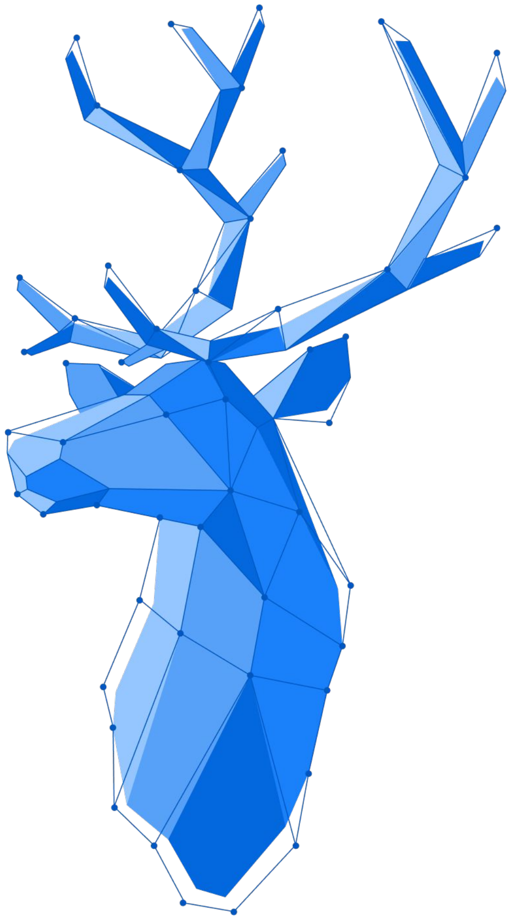
Creating Forms: connecting user inputs to data

Vaadin Flow

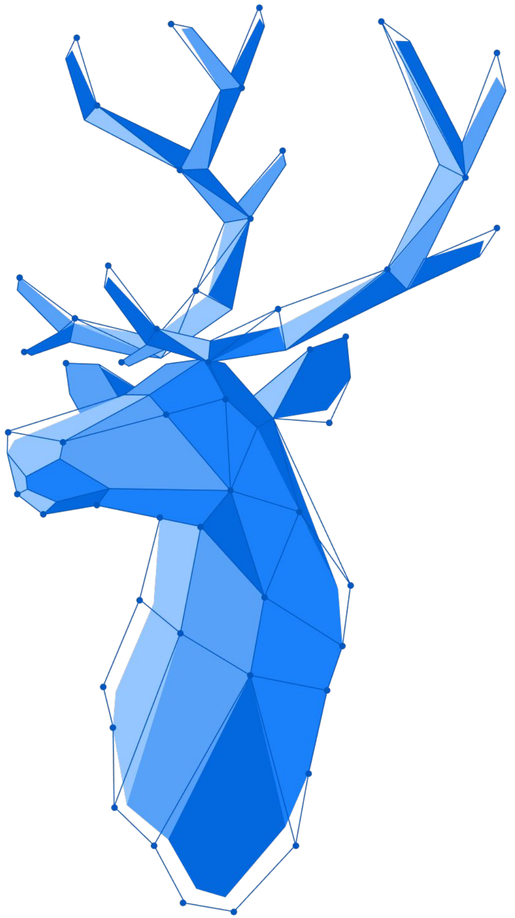


Vaadin training set

Vaadin Foundation

Agenda

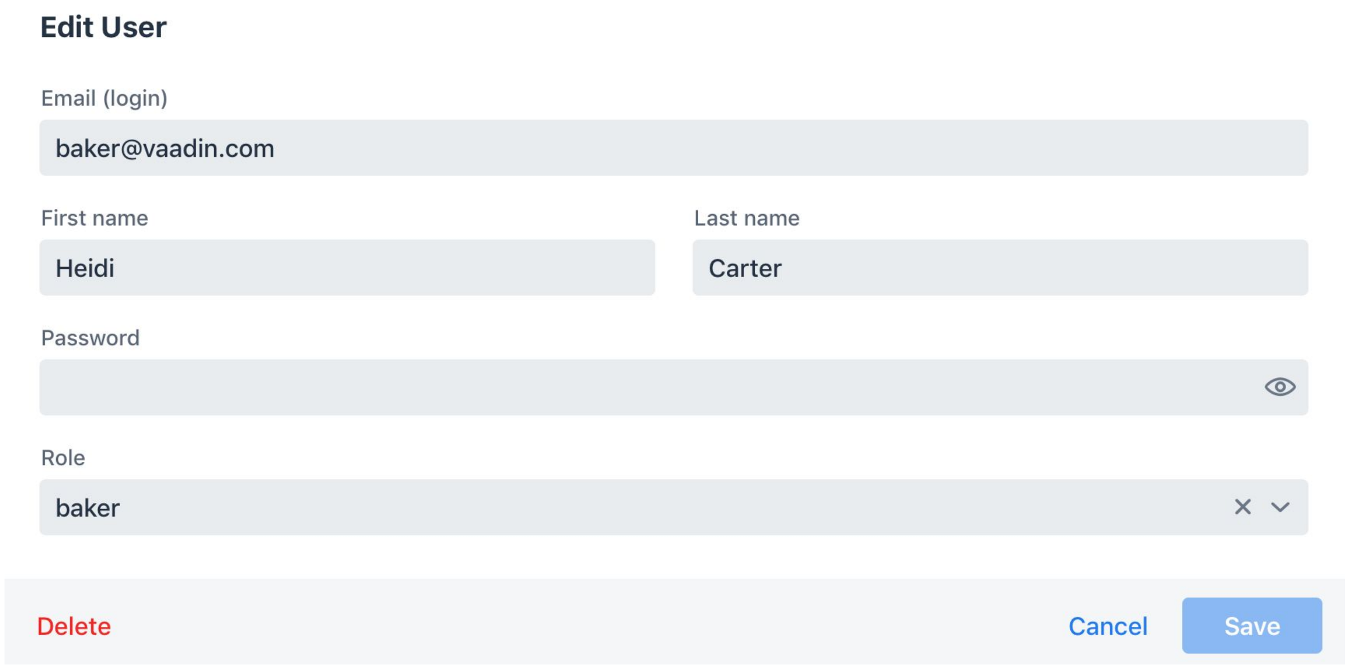
- Part 1:
  - Data Binding in Flow
- Part 2:
  - Validation
  - Conversion
- Part 3:
  - Custom Field



Forms and Data Binding, Part 1

Data binding in Flow

This is a form

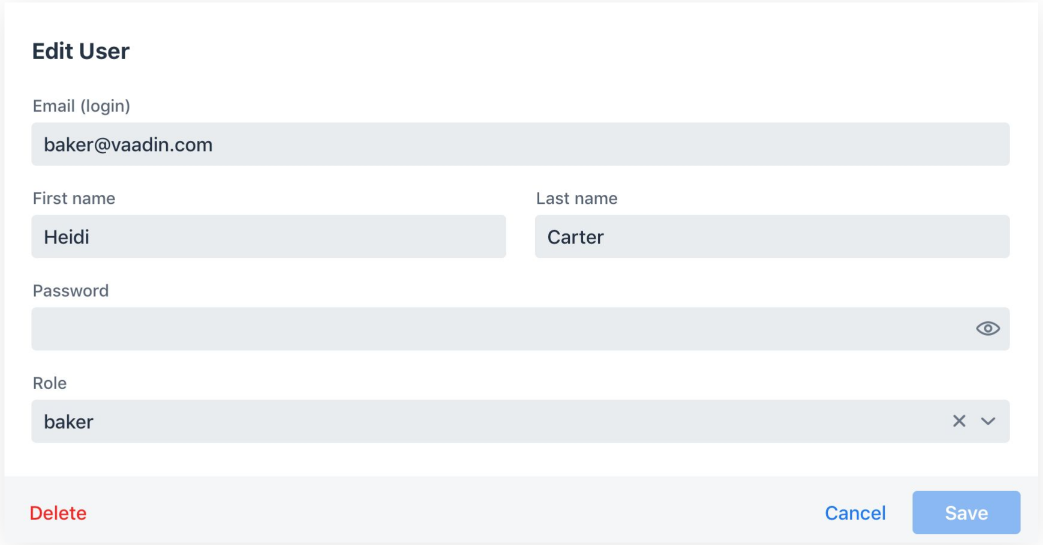


Edit User

Email (login)	baker@vaadin.com
First name	Heidi
Last name	Carter
Password	
Role	baker

How to connect data to the form?

```java
public class User extends AbstractEntity {

  private String email;

  private String passwordHash;

  private String firstName;

  private String lastName;

  private String role;

  …
  // Properties need getters and setter
}
```


Edit User

Email (login): baker@vaadin.com

First name: Heidi

Last name: Carter

Password:

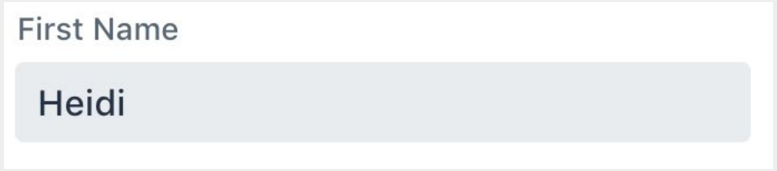
Role: baker

Vocabulary

Fields

A **Field** is a Vaadin component that holds a **value**, for example the TextField:

```
class TextField extends Component
        implements HasValue
```



Vocabulary

Properties

A JavaBean **Property**, like **email** for the User class this example, is a named attribute that can be accessed by a public getter and a setter.

The naming of the methods must follow the JavaBean specification:

- "get" + property name and "set" + property name
- Special case for getters of boolean type properties: "is" + name, like "`boolean isRunning()`"

```
public class User extends AbstractEntity {

  private String email;

  public String getEmail() {
    return email;
  }

  public void setEmail(String email) {
    this.email = email;
  }

  ...
```

All Fields implement HasValue<E, V>

V specifies the type of the data Value La V es la más importante de las dos

- String for TextField, Integer for IntegerField...

E specifies the type of the value change Event

- More about this soon

HasValue<E, V> has 3 methods:

```
V getValue()
```

```
void setValue(V value)
```

```
Registration addValueChangeListener(ValueChangeListener<? super E> listener)
```

Value change events

Es la E de HasValue<E, V>

A Field’s value can change in two different ways:

- Calling setValue() in Java code directly or indirectly
- The user changes the value on the client side

You can react to the changing value using a **value change listener**:

```
TextField textField = new TextField();

textField.addValueChangeListener(event -> {
   if(event.isFromClient()) {
       System.out.println(event.getValue());
   }
});
```

Truco para saber si ha sido el usuario el que ha cambiado el valor

Data Binding

Data Binding with Binder

Vaadin’s data binding system is designed for **reading and writing application data** from UI components

Data can be bound to individual fields or forms containing multiple fields, and to listing components. Listing components are discussed separately, as they have their own API. (como grids)

At the core of the form data binding is a helper class called **Binder**, which takes care of reading values from the business object(s) and showing them in field Components

Binder

Binder binds the business data to field components (anything that implements HasValue)

- Handles data conversion and validation
- Like HasValue, the Binder is always typed to the backing bean:

  Binder<Person> binds fields to a Person object

Initializing Binder

You can instantiate Binder with or without class parameter

```
//Option 1: Doesn't support binding with property names.
Binder<Person> binder = new Binder<>();

//Option 2: Scan class for properties so you can bind by String property name.
Binder<Person> binder = new Binder<>(Person.class)
```

Creating Bindings with Binder

Creating Bindings

A **Binding** describes how to move data between a Field and a data object’s Property

A shared Binder is used to create bindings between each Property in a bean class and its corresponding Field:

```
Binder<Person> binder = new Binder<>();

TextField titleField = new TextField();                 // a Person’s title
IntegerField birthYearField = new IntegerField();       // a Person’s birth year

binder.forField(titleField)...                          // create a binding for titleField
binder.forField(birthYearField)...                      // create a binding for birthYearField
```

There are multiple ways to create Bindings, both manual and automatic.

Bindings with method references

La forma más simple

```
Binder<Person> binder = new Binder<>();
TextField titleField = new TextField();

binder.forField(titleField)            // Start by defining the Field instance to use
       .bind(                          // Finalize by doing the actual binding to the Person class
               Person::getTitle,       // Callback that loads the title from a person instance
               Person::setTitle));     // Callback that saves the title in a person instance
```

Bindings with the property name

```
Binder<Person> binder = new Binder<>(Person.class);
TextField titleField = new TextField();

binder.forField(titleField).bind("title");       // Alternative bind method for Java beans
```

Requiere pasar la clase del objeto como parámetro al constructor del Binder

Shorthands for creating bindings

```
Binder<Person> binder = new Binder<>(Person.class);
TextField titleField = new TextField();

// Shorthand for cases without extra configuration
binder.bind(titleField, Person::getTitle, Person::setTitle);
```

Esta forma es type safe

```
// or with property name
binder.bind(titleField, "title");
```

En tiempo de compilación no sabemos si la propiedad “title” existe. No es type safe

No siempre es posible usar la forma corta. Usando esta forma no pueden añadirse validadores ni conversores

Binding with a lambda expression

```java
Binder<Person> binder = new Binder<>(Person.class);
TextField titleField = new TextField();

// With lambda expressions
binder.bind(titleField,
       person -> person.getTitle(),
       (person, newTitle) -> person.setTitle(newTitle));
```

Otra forma de crear bindings

Bindings with an anonymous class

```java
Binder<Person> binder = new Binder<>(Person.class);
TextField titleField = new TextField();

// With explicit callback interface instances (Java 7 style)
binder.bind(titleField,
       new ValueProvider<Person, String>() {
           @Override
           public String apply(Person person) {
               return person.getTitle();
           }
       },
       new Setter<Person, String>() {
           @Override
           public void accept(Person person, String title) {
               person.setTitle(title);
           }
       });
```

Otra forma más compleja (más explícita) de crear bindings

Automatic binding: bindInstanceFields

Automatic binding uses the field variable name or the @PropertyId annotation

```java
public class MyForm {
   private TextField name = new TextField();

   @PropertyId("email")
   private TextField emailField = new TextField();

   public MyForm() {
       Binder<Person> binder = new Binder<>(Person.class);
       binder.bindInstanceFields(this);
       // name is now bound to “name”, emailField is bound to “email”
   }
}
```

```java
public class Person {
   private String name;
   private String email;

   // getter and setter...
}
```

Necesario pasar el nombre de la clase como parámetro al constructor. Muy útil para formularios muy grandes

Bindings for nested properties

What if we have a complex object with other beans as its Properties?

```
public class Person {

    private String name;
    private Address address;

    ...
}

public class Address {

    private String street;
    ...
}
```

Bind nested properties with Strings

Bind nested properties with the "`property.subproperty`" syntax

```
Binder<Person> binder = new Binder<>(Person.class);

TextField streetField = new TextField();

// this works if you can guarantee "address" is not null
binder.forField(streetField).bind("address.street");
```

```
public class Person {

    private String name;
    private Address address;

    ...
}

public class Address {

    private String street;
    ...
}
```

Necesario pasar el nombre de la clase como parámetro al constructor. Si address es null, obtenemos el error NullPointerException

Bind nested properties with a lambda

```
Binder<Person> binder = new Binder<>(Person.class);

TextField streetField = new TextField();

binder.forField(streetField).bind(
       person -> person.getAddress().getStreet(),
       (person, street) -> person.getAddress().setStreet(street));

// Still not safe if getAddress() can return null
```

```
public class Person {

    private String name;
    private Address address;

    ...
}

public class Address {

    private String street;
    ...
}
```

Bind nested properties with a lambda

Bind nested properties with the a lambda expression and a null check

```
Binder<Person> binder = new Binder<>(Person.class);

TextField streetField = new TextField();

binder.forField(streetField).bind(
        person -> {
            if(person.getAddress()==null){
                return null;
            }else{
                return person.getAddress().getStreet();
            }
        },
        (person, street) -> {
            if(person.getAddress()!=null){
                person.getAddress().setStreet(street);
            }
            // should we create a new Address object otherwise?
        });
```

```
public class Person {

    private String name;
    private Address address;

    ...
}

public class Address {

    private String street;
    ...
}
```

Dependerá de la lógica de negocio

Reading and writing with Binder

Buffered or immediate

Cómo ocurre la lectura y la escritura con Binder: immediate y buffered

Immediate reading and writing values

You can use *unbuffered* binding by calling binder.setBean. This means changes to the fields are updating the bound properties immediately. Fields are also updated with initial values when **setBean** is called.

```
Person person = getPerson();
Binder<Person> binder = new Binder<>();


binder.setBean(person);     // Sets the person instance as a data source for the binder
                             // Bound fields are updated, and the bean is updated when fields change
```

No se usa mucho

Buffered reading and writing values

Using a binder’s **readBean** reads the values once and does not hold a reference to the bean. Field values are only committed back to the properties when a binder.**writeBean** call is made.

```java
person = getPerson();
Binder<Person> binder = new Binder<>();

// Buffered binding.
binder.readBean(person);  // Reads values from the Person instance to the binder and updates fields
Button saveButton = new Button("Save", event -> {
   try {
       binder.writeBean(person);   // Writes values from the binder to the person object
       // After successful writeBean, binder.hasChanges() == false
   } catch (ValidationException e) {
       // Could not save the values; check exceptions for each bound field
   }
});
```

La forma más normal de usarlo, con interacción del usuario, por ejemplo al pulsar un botón…

Writing as draft

Binder.writeBeanAsDraft(bean) allows you to write out changes despite some fields not passing validation - this is useful when you want to allow saving of an incomplete form (“save and resume later”). We’ll return to writing as draft after discussing validation and conversion later.

```
Button saveButton = new Button("Save draft", event -> {
       // Write the changes that are valid
       binder.writeBeanAsDraft(person);
});

Button saveEvenInvalidButton = new Button("Resume later", event -> {
        // Write all changes that pass conversion - skips validation!
        // Note: If the conversion fails, the value written to the bean will be null.
       binder.writeBeanAsDraft(person, true);
});
```

Writing as draft doesn’t reset the changed bindings list - binder.hasChanges() still returns “true” if changes occurred before writing.

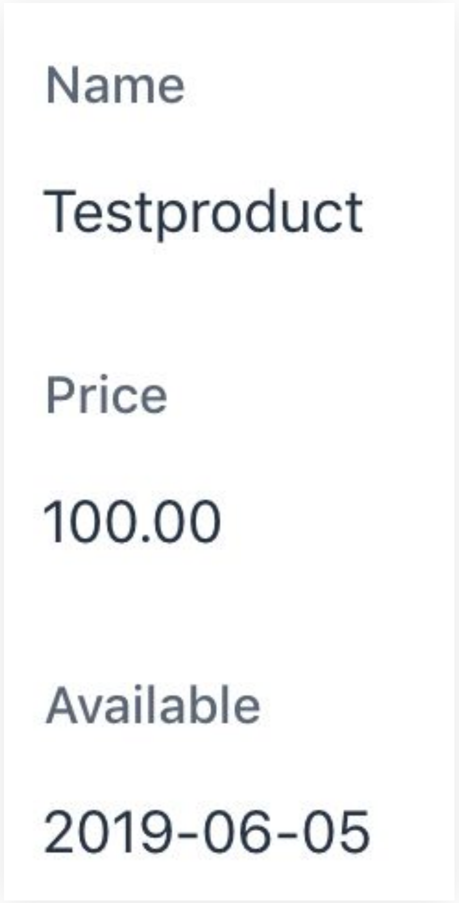
Reset buffered value

Buffered writing allows you to cancel edits by calling binder.readBean again instead of writeBean

```
person = getPerson(); // person is defined as a class member variable
Binder<Person> binder = new Binder<>();

// Buffered binding.
binder.readBean(person);    // Reads values from the Person instance to the binder
Button saveButton = new Button("Save", event -> {
   try {
       binder.writeBean(person);   // Writes values from the binder to the person object
   } catch (ValidationException e) {
       // Could not save the values; check exceptions for each bound field
   }
});
Button cancelButton = new Button("Cancel", event ->
       binder.readBean(person)
       // person has not been updated before writeBean -> revert changes in Fields back
);
```

Binding to a non-Field

Sometimes you might need to bind data to a non-field Component, like a Span or a Paragraph, to make the data read-only.



Name

Testproduct

Price

100.00

Available

2019-06-05

Binding with ReadOnlyHasValue

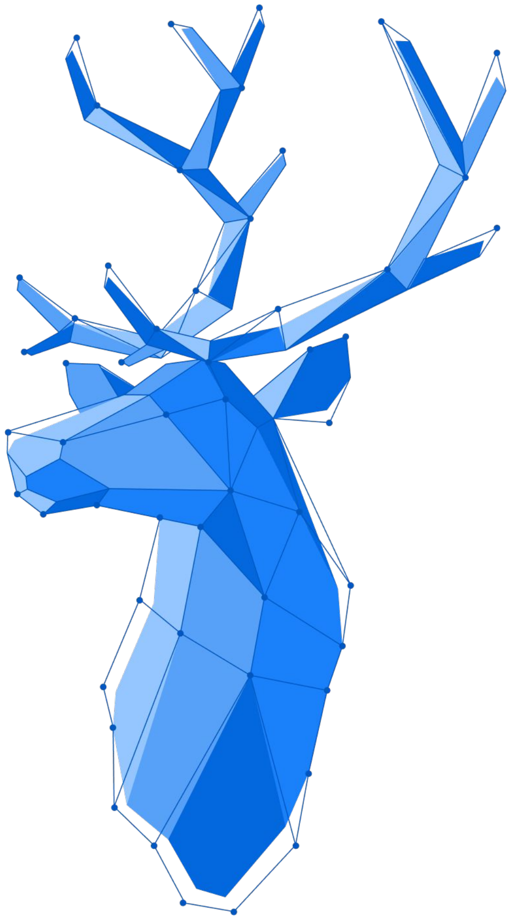
The method bindReadOnly also works for normal Fields.

```
Div nameText = new Div();
ReadOnlyHasValue<String> name = new ReadOnlyHasValue<>(text -> nameText.setText(text));
binder.forField(name).bindReadOnly(Person::getName);
// or the same thing:
binder.forField(name).bind(Person::getName, null);
```

setter a null

Summary, Part 1

- Fields and Properties
- Binder
- Creating Bindings
- Reading and writing values with Bunder
  - Buffered (immediate)
  - Unbuffered

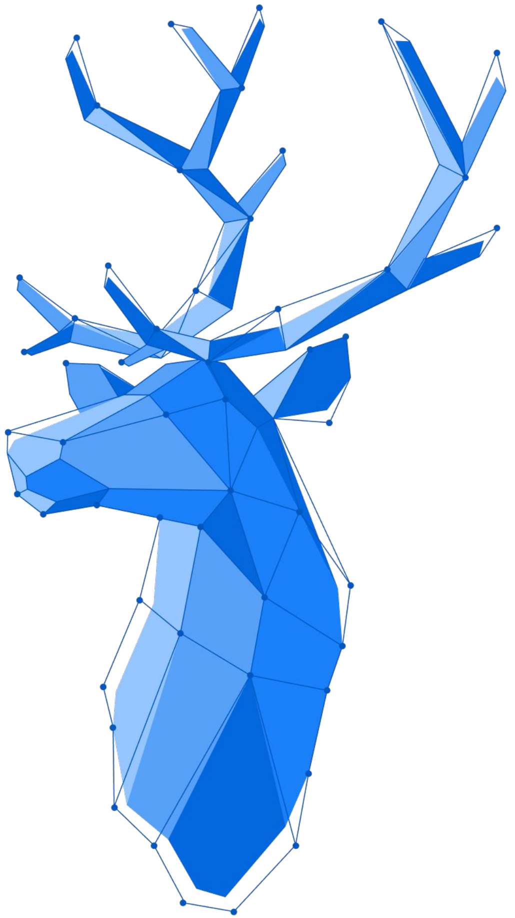


Creating Forms, Part 2

Validation and Conversion

Recap, part 1

- Fields and Properties
- Creating Bindings to Components with Binder
- Reading and writing data

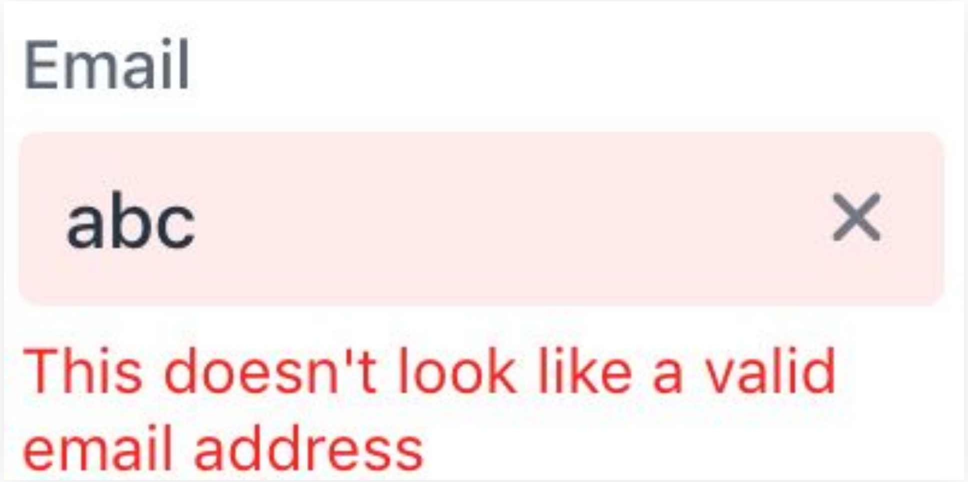


Validation

Validation

Validation is used to check that a field’s value satisfies defined formats and other input criteria.

When validation fails, a field will turn red and display an error message next to it.


Email

abc

This doesn't look like a valid email address

Validation with a Validator

Use .withValidator to configure a Validator for the binding you start with binder.forField

```
binder.forField(emailField)
        // Explicit Validator instance
        .withValidator(new EmailValidator("This doesn't look like a valid email address"))
        .bind(Person::getEmail, Person::setEmail);
```

Validator es el interface que Vaadin proporciona, y .withValidator() el método

Validation with a Validator

Use .withValidator to configure a Validator for the binding you start with binder.forField

```
binder.forField(emailField)
        // Explicit Validator instance
        .withValidator(new EmailValidator("This doesn't look like a valid email address"))
        .bind(Person::getEmail, Person::setEmail);
```

Notes:

- The binder.forField method call starts a **builder** where the field definition is followed by any number of configuration calls and the `.bind` call finalizes the chain.
  - You can specify multiple validators by chaining them!
- The shorthand method `binder.bind(emailField, Person::getEmail, Person::setEmail)` doesn't allow setting Validators

Validation with a lambda

You can specify your custom inline implementation for a validator easily with a lambda expression and a String error message:

```
binder.forField(nameField)
       // Validator defined based on a lambda and an error message
       .withValidator(
               name -> name.length() >= 3, "Full name must contain at least three characters")
       .bind(Person::getName, Person::setName);
```

The asRequired shorthand

The asRequired shorthand binding method allows for a convenient “not null / empty” validation check, and it also enables the Field’s required indicator

```
binder.forField(titleField)
        // Shorthand for requiring the field to be non-empty
        .asRequired("Every employee must have a title")
        .bind(Person::getTitle, Person::setTitle);
```

The Validator API

The Validator interface works with two helper interfaces: ValueContext and ValidationResult

```java
public interface Validator<T>{

   @Override
   ValidationResult apply(T value, ValueContext context);

}
```

ValidationResult

ValidationResult is an interface; you can implement your own subclass if you want to pass additional information. The ValidationResult's “isError” method determines whether the validation passes or fails.

```
public class MyValidator implements Validator<String> {

   @Override
   public ValidationResult apply(String value, ValueContext context){
       if(value == null || value.length() < 3) {
           return ValidationResult.error(“String is too short”);
       } else {
           return ValidationResult.ok();
       }
   }
}
```

Los valores por defecto son ValidationResult.ok() y ValidationResult.error(), pero se puede implementar como lo necesitemos

ValueContext

The ValueContext object that comes as an input to the apply method of the Validator interface is populated by the framework. It contains information about the binding target and current Locale.

```
public class ValueContext implements Serializable {

  private final Component component;
  private final HasValue<?, ?> hasValue;
  private final Locale locale;
  ...
```

Nos puede servir para saber si los decimales son puntos o comas…

Built-in validators

- BeanValidator
- BigDecimalRangeValidator
- BigIntegerRangeValidator
- ByteRangeValidator
- DateRangeValidator
- DateTimeRangeValidator
- DoubleRangeValidator
- EmailValidator
- FloatRangeValidator
- IntegerRangeValidator
- LongRangeValidator
- RangeValidator
- RegexpValidator
- ShortRangeValidator
- StringLengthValidator

Custom validation result handling

You can direct Binder to use a specific component to display any error messages. By default, the error message is displayed next to the Field.

```
Paragraph errorHolder = new Paragraph();

binder.forField(nameField).asRequired().withStatusLabel(errorHolder).bind("name");

binder.forField(emailField).asRequired().withStatusLabel(errorHolder).bind("email");
```

Fully customized validation error handler

You can have even more control over the validation error handling by using .withValidationStatusHandler

```
binder.forField(nameField).asRequired().withValidationStatusHandler(status -> {
    // do something
}).bind("name");

binder.forField(emailField).asRequired().withValidationStatusHandler(status -> {
    // do something
}).bind("email");
```

Cuando withStatusLabel no es suficiente. Se pueden hacer error handlers más complejos, como notificaciones, o limpiar el mensaje de error cuando la validación es correcta

Binder-level validation result handling

Instead of adding status label or status handler to each field, you can also add one to the Binder

```
binder.setStatusLabel(errMsg);

binder.setValidationStatusHandler(status -> {
    // do something for every validation status change
});
```

Binder-level validator

Add validator to binder directly to do cross-field validation. Binder-level validation will only run if field level validation has passed.

```
/*
* Note that it won't automatically make any fields invalid or show the error message if it fails.
* Combine it with binder.setStatusLabel() and binder.setValidationStatusHandler() to show the error message and invalidate the fields.
*/
binder.withValidator(flight ->
                flight.getDepartureDate().isBefore(flight.getReturnDate())
                        || flight.getDepartureDate().isEqual(flight.getReturnDate()),
        "Please select a valid date range");
```

Bean Validation (JSR-303)

Use the BeanValidationBinder for Bean validation. Validation messages are read from ValidationMessages.properties resource bundle file

```
public class Product {

  @NotBlank(message = "{bakery.name.required}")
  @Size(max = 255, message = "{bakery.field.max.length}")
  private String name;

}


TextField name = new TextField();
BeanValidationBinder<Product> binder = new BeanValidationBinder<>(Product.class);

binder.bind(nameField, "name");
```

Hay que pasar la clase

No puede usarse method reference ni lambda expressions. Hay que usar el nombre de la propiedad

Bean Validation

To use Bean Validation, you can ONLY do the data binding with property name.

Bean Validation

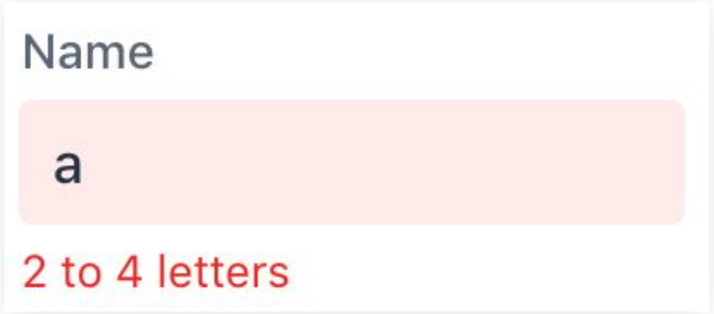
Bean validation needs the validation API and an implementation in your dependencies

```
<!-- API -->
<dependency>
    <groupId>jakarta.validation</groupId>
    <artifactId>jakarta.validation-api</artifactId>
    <version>3.0.2</version>
</dependency>

<!-- One possible implementation -->
<dependency>
   <groupId>org.hibernate.validator</groupId>
   <artifactId>hibernate-validator</artifactId>
   <version>6.0.2.Final</version>
</dependency>
```

Client-side validation

Components allow you to set validations directly on the field, not through a binder.

```
TextField name = new TextField("Name");
name.setRequired(true);
name.setMinLength(2);
name.setMaxLength(4);
name.setErrorMessage("2 to 4 letters");
```


Name

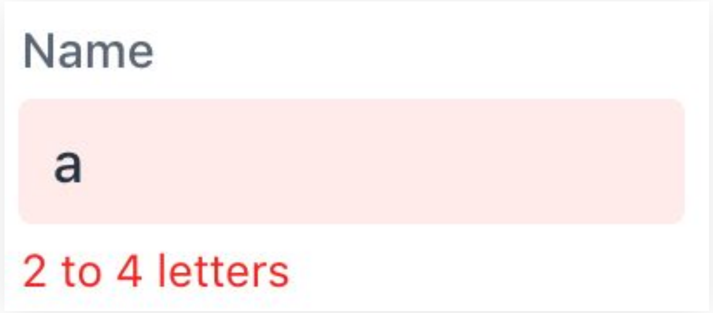
a

2 to 4 letters

Estas validaciones ocurren en el navegador, no en el servidor

Client-side validation

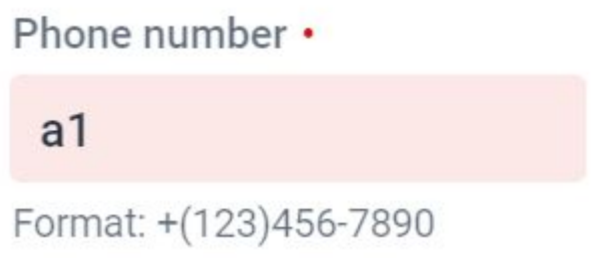
Client-side validations aren’t secure against tampering and shouldn’t be relied on alone.



Client-side constraints

Every field component provides its own set of **constraints**, such as **`required`**, **`minlength`**, **`pattern`**, etc. Component constraints are integrated into Binder validation. Binder checks the value against constraints before custom validators.

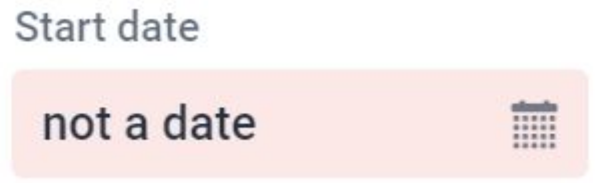
The only exception is the **`required`** constraint, which is checked with the **`asRequired`** validator in Binder.



Client-side to server validation sync

Certain components, such as IntegerField or DatePicker, only accept input that can be parsed as a suitable **type** (`Integer`, `LocalDate`, etc.). Otherwise, the value on the server falls back to null.

The input type check, as well as all other validations, happen on **blur** - when the field loses focus



Conversion

Conversion

Both Fields and Bindings are typed.

When types don’t match, you need a converter.

No conversion?

- When you’re binding with a String property name, you may get a **runtime exception** if the type of the Field and the type of the Property do not match.
- If you create the Binding with lambdas, method references or classes, your binding code will not **compile**.
    - You will catch type errors earlier

```
public class Person {
   private LocalDate birthDate;
}
```

```
TextField birthDateField = new TextField("Birth date");
Binder<Person> binder =
       new Binder<>(Person.class);
binder.bind(birthDateField, "birthDate");
```

```
Caused by: java.lang.ClassCastException:
java.base/java.lang.String cannot be cast to
java.base/java.time.LocalDate
       at
com.vaadin.flow.component.textfield.TextField.setValue(Tex
tField.java:32)
       at
com.vaadin.flow.data.binder.Binder$BindingImpl.initFieldVa
lue(Binder.java:1130)
       at ...
```

Converter

Add a Converter before binding:

```
TextField age = new TextField("Age");
Binder<Person> binder = new Binder<>(Person.class);

binder.forField(age)
       .withConverter(new StringToIntegerConverter("Must enter a number"))
       .bind("age");
```

Converter API

The Converter interface uses the ValueContext and Result helpers. Conversion is done between a **presentation** type (what the Field uses) and a **model** type (what the Property has).

```
public class MyStringToDoubleConverter implements Converter<String, Double> {

   @Override
   public String convertToPresentation(Double value, ValueContext context) {
       return String.format(context.getLocale().get(), "%1$.2f", value);
   }

   @Override
   public Result<Double> convertToModel(String value, ValueContext context) {
       try {
           return Result.ok(Double.parseDouble(value));
       }
       catch (NumberFormatException ex) {
           return Result.error(ex.getMessage());
       }

   }
}
```

Built-in converters

DateToLongConverter
DateToSqlDateConverter
LocalDateTimeToDateConverter
LocalDateToDateConverter
StringToBigDecimalConverter
StringToBigIntegerConverter
StringToBooleanConverter
StringToDateConverter
StringToDoubleConverter
StringToFloatConverter
StringToIntegerConverter
StringToLongConverter
StringToUuidConverter

Automatic conversion

If a Converter isn't specified, and the types between the Field and the Property do not match, there's a call to the default converter factory that attempts to find a mapping using the built-in converters.

You can also provide your own implementation of the ConverterFactory interface by extending Binder and overriding `getConverterFactory`.

Converters and Validators

You can combine multiple validators and converters. They will be executed in the same order as they are defined.

```
binder.forField(age)
       .withValidator(value -> validateIsNumber(value), "You can only enter numbers!")
       .withConverter(new StringToIntegerConverter("Not a valid integer!"))
       .withValidator(integer -> isAdult(integer), "Must be 18 or older!")
       .bind("age");
```

Writing as draft

Binder.writeBeanAsDraft(bean) allows you to write out changes despite some fields not passing validation - this is useful when you want to allow saving of an incomplete form (“save and resume later”)

```
Person person = getPerson();
Binder<Person> binder = new Binder<>();
binder.forField(field1).withValidator(...
binder.forField(field2).withValidator(...

Button saveButton = new Button("Save draft", event -> {
       // Write the changes that are valid
       binder.writeBeanAsDraft(person);
        // Note: binder.hasChanges is still true, Binder’s list of changed bindings is not reset
});
```

Opción 1

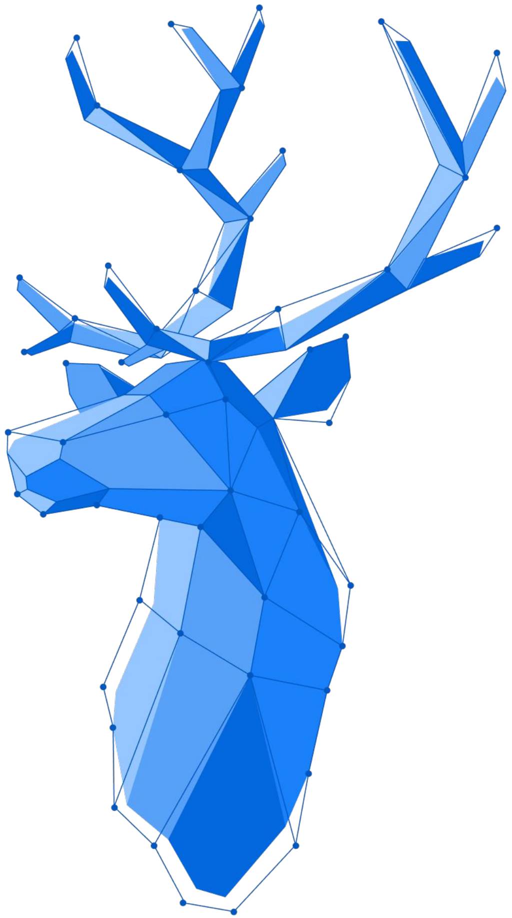
```
Button saveEvenInvalidButton = new Button("Resume later", event -> {
        // Write all changes that pass conversion - skips validation!
        // Note: If the conversion fails, the value written to the bean will be null.
       binder.writeBeanAsDraft(person, true);
});
```

Opción 2

Exercise 1

Summary, Part 2

- Validation
- The Validator API
- Validation result handling
- Bean validation
- Exercise 1
- Conversion
- The Converter API
- Writing as draft

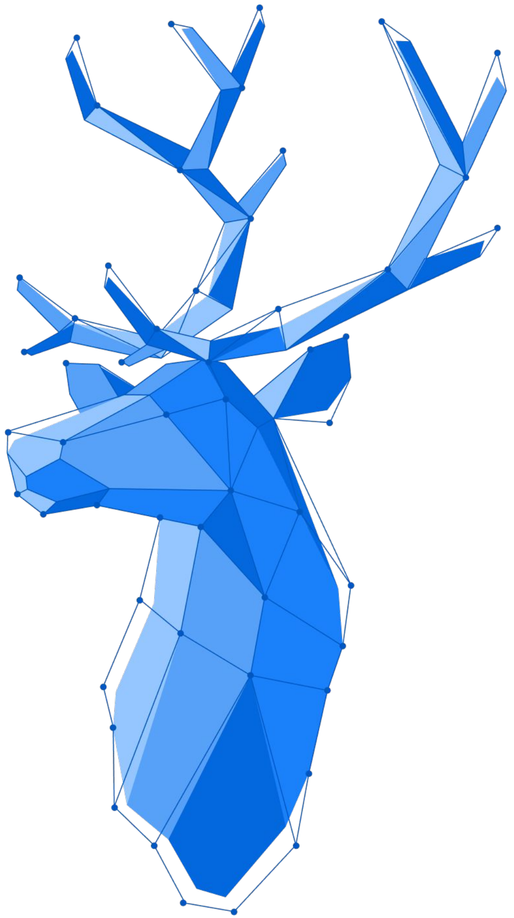


Creating Forms, Part 3

Custom Field

Recap, parts 1–2

- Fields and Properties
- Creating Bindings to Components with Binder
- Reading and writing data
- Validation
- Conversion



Custom Field

Custom Fields

You are not limited to use only Vaadin components as Fields, you can use a custom web component as well.

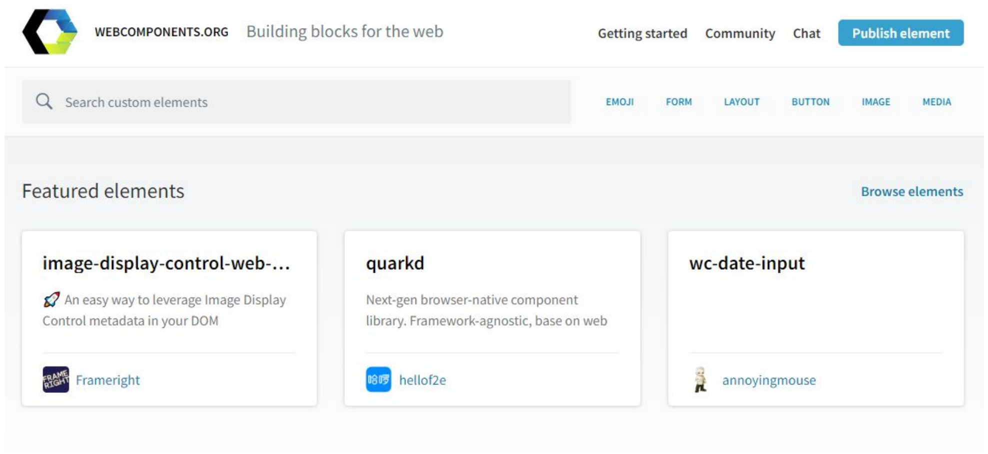


image-display-control-web-...

🚀 An easy way to leverage Image Display Control metadata in your DOM

Frameright

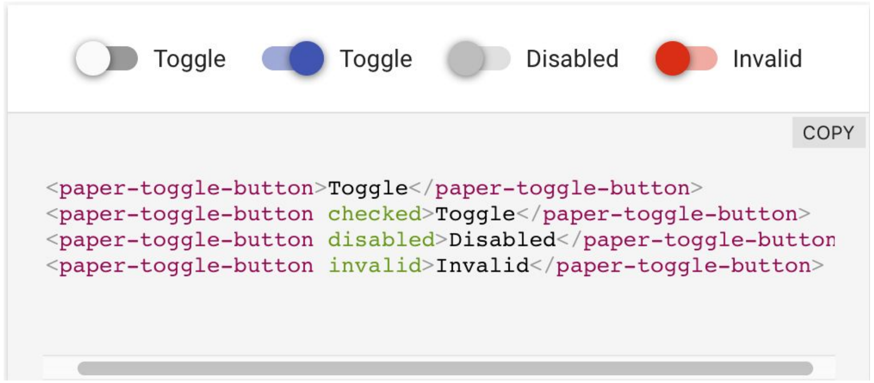
quarkd

Next-gen browser-native component library. Framework-agnostic, base on web

hellof2e

wc-date-input

annoyingmouse

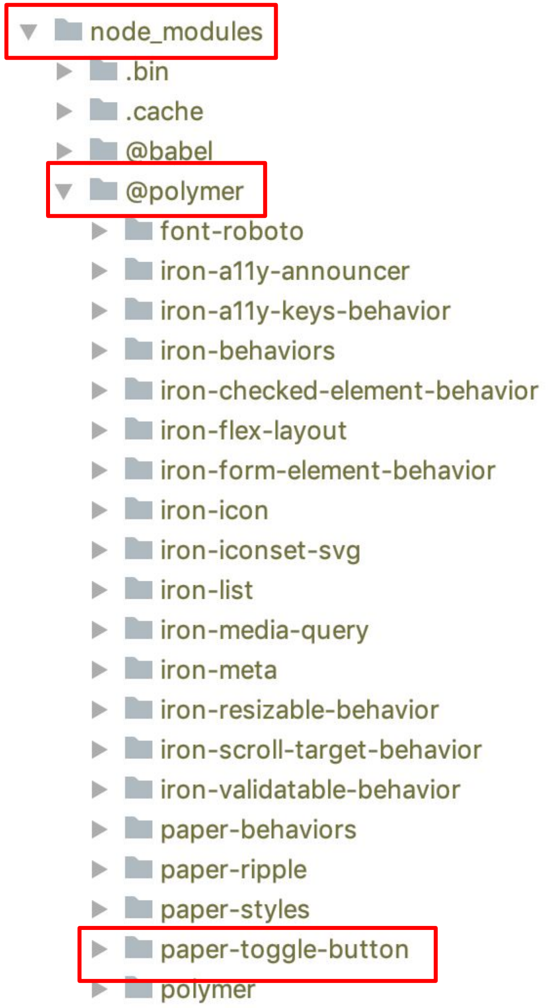


```
<paper-toggle-button>Toggle</paper-toggle-button>
<paper-toggle-button checked>Toggle</paper-toggle-button>
<paper-toggle-button disabled>Disabled</paper-toggle-button
<paper-toggle-button invalid>Invalid</paper-toggle-button>
```

Install

Run npm install on the project's root directory

```
npm install --save @polymer/paper-toggle-button
```



```
▼ node_modules
  ▶ .bin
  ▶ .cache
  ▶ @babel
  ▼ @polymer
    ▶ font-roboto
    ▶ iron-a11y-announcer
    ▶ iron-a11y-keys-behavior
    ▶ iron-behaviors
    ▶ iron-checked-element-behavior
    ▶ iron-flex-layout
    ▶ iron-form-element-behavior
    ▶ iron-icon
    ▶ iron-iconset-svg
    ▶ iron-list
    ▶ iron-media-query
    ▶ iron-meta
    ▶ iron-resizable-behavior
    ▶ iron-scroll-target-behavior
    ▶ iron-validatable-behavior
    ▶ paper-behaviors
    ▶ paper-ripple
    ▶ paper-styles
    ▶ paper-toggle-button
    ▶ polymer
```

Web Component as a Field

Many field-type web components usually have a property which holds the value. You can extend AbstractSinglePropertyField in a custom class to turn such a web component into a Vaadin-compatible Field.

AbstractSinglePropertyField takes two generic parameters. The first one is the Component type and the second one is the value type.

Usually only a super constructor call with 3 parameters is needed. The first is the property name which holds the value, the second one is the default value, and the third one defines if null is allowed.

```java
@JsModule("@polymer/paper-toggle-button/paper-toggle-button.js")
@Tag("paper-toggle-button")
public class ToggleButton extends AbstractSinglePropertyField<ToggleButton, Boolean> {
   public ToggleButton() {
      // property name, default value, accept null values
      super("checked", false, false);
   }
}
```

Limitation of AbstractSinglePropertyField

AbstractSinglePropertyField holds the value properly, but it doesn't have validation error message or required indicator.

CustomField

CustomField is a wrapper of component(s) to be used as a regular field.

CustomField has a validation error message and a required indicator.

Making a CustomField

Extend from CustomField which takes a generic parameter for the data type

```java
public class MyCustomField extends CustomField<Boolean> {
```

Set up the content of the CustomField

Here we'll use the web component field as the inside implementation

```java
public class MyCustomField extends CustomField<Boolean> {

   private ToggleButton toggleButton = new ToggleButton();

   public MyCustomField(){
       add(toggleButton);
   }
```

Implement value methods

Next, implement the generateModelValue() and setPresentationValue() methods

```
public class MyCustomField extends CustomField<Boolean> {

   private ToggleButton toggleButton = new ToggleButton();

   public MyCustomField(){
       add(toggleButton);
   }
   @Override
   protected Boolean generateModelValue() {
       return toggleButton.getValue();
   }

   @Override
   protected void setPresentationValue(Boolean newPresentationValue) {
       toggleButton.setValue(newPresentationValue);
   }
}
```

Combine multiple components

You can also combine server-side Field components into a single CustomField

```java
public class MyCustomField extends CustomField<LocalDateTime> {

   private DatePicker datePicker = new DatePicker();
   private TimePicker timePicker = new TimePicker();

   public MyCustomField(){
       add(datePicker, timePicker);
   }

   @Override
   protected LocalDateTime generateModelValue() {
       return LocalDateTime.of(datePicker.getValue(), timePicker.getValue());
   }

   @Override
   protected void setPresentationValue(LocalDateTime newPresentationValue) {
       datePicker.setValue(newPresentationValue.toLocalDate());
       timePicker.setValue(newPresentationValue.toLocalTime());
   }

}
```

Upload as a Field

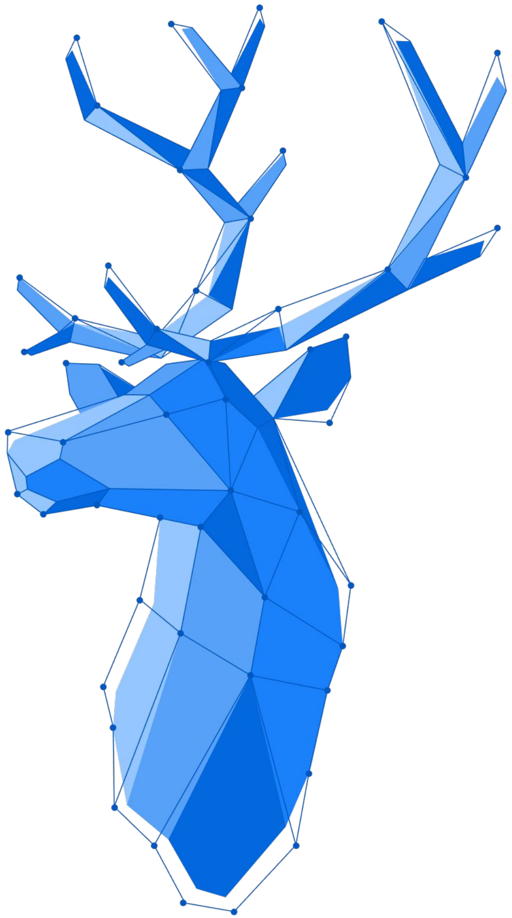
The Upload component doesn’t implement HasValue, so it can’t be used with Binder directly. You can get around this by implementing a CustomField that wraps Upload and handle the file data there:

```java
public class UploadField extends CustomField<InputStream> {
  InputStream is;
  FileBuffer buffer = new FileBuffer(); // You might want to customize this

  public UploadField() {
    Upload upload = new Upload(buffer);
    upload.setMaxFiles(1);
    upload.addSucceededListener(event -> {
      is = buffer.getInputStream();
    });
    add(upload);
  }

  @Override
  protected InputStream generateModelValue() {
    return is;
  }
  @Override
  protected void setPresentationValue(InputStream newPresentationValue) {
  }
```

Exercise 2

Summary, Part 3

- Custom Fields
- Web Component as a Field
- AbstractSinglePropertyField
- CustomField
- Exercise 2



Thank you!