supler Documentation

Release 1.0

Softwaremill

November 10, 2015

Contents

Supler is an open-source library which makes writing complex forms easier. It has a server-side (Scala) and a client-side (JavaScript) component.

Writing your first Supler form is just *a link away*!

On the server side Supler provides:

- a DSL for defining forms
- generating a JSON description of a form, reading values from a data object
- applying JSON, writing to a data object
- running server-side conversions and validations
- running server-side actions

On the frontend side Supler provides:

- generating HTML basing on JSON form description
- serializing a (possibly modified) form to JSON
- running client-side validations
- customization of the HTML generation process
- automatically refreshing the form after a field is changed or an action invoked

As important as Supler's features, are things that Supler **does not** do. Supler does not define or mandate how the objects backing the forms should work, how are they persisted, what is their lifecycle; it is agnostic as to which Javascript/web framework you use or how HTTP sessions are managed. The generated HTML has elements with predictable names and can be easily customized.

License: Apache2.

Head over to our user/development forum if you have any questions.

CHAPTER 1

Complete documentation

1.1 Your first Supler form

1.1.1 Server-side

Let's say we have the following class on the backend:

```
case class Person(name: String, age: Int, address: Option[String], likesChocolate: Boolean)
```

A form backed by instances of the `Person` class for editing a user's details can then have the following definition:

```
import org.supler.Supler._

val personForm = form[Person](f => List(
  f.field(_.name).label("Name"),
  f.field(_.age).label("Age").validate(gt(1), le(150)),
  f.field(_.address).label("Address"),
  f.field(_.likesChocolate).label("Do you like chocolate?"),
  f.action("save") { p =>
    println("Saving person: " + p)
    ActionResult.custom(JString("Saved"))
  }.label("Save").validateAll()
))
```

As you can see, a form contains a list of fields. Here we are using a convenience function, `form`, to capture the class of the object once, so that we can later specify the fields using closures like: `_.fieldName`, without any type annotations.

Each field can be customized. For example, here every field has a label. Also, the `age` field has two validators: age must be > 1 and <= 150 (we are optimistic about the development of education and medicine). All fields which are not wrapped in an `Option` have also a "required" validation added.

Finally, besides the four fields, we have an action, which - as the label probably reveals - is supposed to save the form. Here we just print the object to the console. How you handle actions, and how you actually "save" objects is up to you, all you need to do is provide a closure accepting the modified data. We also specify that to invoke the action, the whole form must be valid (using `validateAll()`).

The result of the save action is a custom JSON. Supler uses json4s for parsing & creating JSON with the native backend; in fact, it is the only dependency of Supler. The custom JSON in this case is just a string message, "Saved" (presumably to be displayed to the user on the frontend).

Having the form defined, we need to expose it to the world, which is typically done using some web or REST framework. You can use servlets with JAX-RS, Spray, Play, Scalatra, or any custom framework you want. To keep the

tutorial simple, let's assume we have two methods mapped to appropriate paths & http methods:

- `getPersonForm(): JValue`, invoked on a `GET /personform`
- `postPersonForm(body: JValue): JValue`, invoked on a `POST /personform` with the body parsed as JSON

The implementation of get is quite straightforward:

```
def getPersonForm(): JValue = {
  val person = lookupCurrentPerson()
  personForm(person).generateJSON
}
```

Again, where do the specific `Person` instances come from, do they come from the session, or if they are looked up basing on HTTP parameters, is outside the scope of Supler.

Once we have a `Person` instance, we apply it to the form (`personForm(person)`), and generate the JSON description of the form, reading field values from the given object (if you are curious how the JSON looks like, check out the *JSON section* of the docs).

The post method is similarly simple:

```
def postPersonForm(body: JValue): JValue = {
  val person = lookupCurrentPerson()
  personForm(person).process(body).generateJSON
}
```

What `process` does is:

- convert and apply values from the given json to the person object
- run validations
- if there are no errors, run the actions (if any)

If no action is invoked (`body` has only field value mappings), this method can be used for doing server-side validation of a form; the result will contain any validation errors, which can be the displayed to the user. In our example, if the `"save"` action was invoked and the data was valid, the result of `process(body).generateJSON` would be the action's result: `JString("Saved")`.

You can also invoke any of the processing steps by hand; this is covered later in the docs.

1.1.2 Frontend

Time to display something! First we need a designated space on our HTML page where the form will be displayed:

```
<html>
<head>
  <script src="/supler.js"></script>
</head>
<body>
  <div id="person_form_container"></div>
</body>
</html>
```

Then, when the page opens, we need to fetch and display the form. I will use JQuery here, but of course any way of doing AJAX calls/networking will work, JQuery is not a dependency of Supler:

```
var form = new Supler.Form(
  document.getElementById('person_form_container'),
  {}
);

$(document).ready(function() {
  $.get('/personform', function(data) {
    form.render(data);
  });
});
```

Here we are creating a `Supler.Form` instance which as the first arguments requires the HTML element where the form should be rendered, and as the second options, which we'll be using shortly. Then, when the document is ready, we are calling the endpoint to get the JSON form description, and we render the results. This will display a 4-field & 1-button form to the user.

What about sending user changes, when an action is invoked (in our case, when the "Save" button is clicked)? Supler doesn't contain any code to e.g. perform AJAX requests. Instead, we need to to provide a way to send data back to the backend. This is configured via the `send_form_function` option, which is an integration point with whatever way of doing network communication we are using:

```
var form = new Supler.Form(
  document.getElementById('person_form_container'),
  {
    send_form_function: sendForm
  })
);

function sendForm(formValue, renderResponseFn, sendErrorFn) {
  $.ajax({
    url: '/personform',
    type: 'POST',
    data: JSON.stringify(formValue),
    dataType: 'json',
    contentType: 'application/json; charset=utf-8',
    success: renderResponseFn,
    error: sendErrorFn
  });
};
```

This is a fairly standard JQuery AJAX call. What is important, is that we are POSTing the form value (received as a parameter) serialized as JSON to the backend, and for handling responses we are using the provided `renderResponseFn` which will re-render the form if there are conversion/validation errors.

After a field is edited, and before an action is invoked, client-side validations are run. If they fail, a message is displayed to the user. Of course, validations are also run on the server, before actually running the action code.

Not all validations are both client- and server-side. Most of the built-in are, but you can also provide custom validations, which are server-side only, or which perform a simplified client-side validation.

Finally, what if an action returns a custom JSON? This must be handled somehow as well. We need to provide a method which will handle such responses:

```
var form = new Supler.Form(
  document.getElementById('person_form_container'),
  {
    send_form_function: sendForm,
    custom_data_handler: function(data) {
      $("#user_feedback").html(data);
```

```
    },
  })
);
```

And that's it! Like on the server-side, you can call any of the stages (serializing a form to JSON, validating, re-rendering with new data) by hand; this will also be covered later in the docs.

1.1.3 What's next?

The various Supler components are described in more detail in further sections. If you'd like to add Supler to your project, head over to *setup*. If you have any questions, feel free to ask on the forum.

1.2 Setup

1.2.1 Stable version

As Supler has two parts, they are deployed in different repositories. The backend can be found in Sonatype's OSS repository, and if you have e.g. an SBT build you just need to add:

```
libraryDependencies += "com.softwaremill.supler" %% "supler" % "0.3.0"
```

The frontend is deployed to Bower, and you can install it simply using `bower install supler`. Or you can just grab `supler.js` directly from the GitHub tag.

1.2.2 Development version

If you like living on the edge, you can use development version of supler.

Add the SNAPSHOT repository (this is SBT, but the repository is maven - you can use it in your favorite build system):

```
resolvers += "OSS JFrog Artifactory" at "http://oss.jfrog.org/artifactory/oss-snapshot-local"
```

and then add the dependencies for both supler backend:

```
libraryDependencies += "com.softwaremill.supler" %% "supler" % "0.4.0-SNAPSHOT"
```

and frontend (in bower.json):

```
"supler": "softwaremill/supler#master"
```

1.3 Live demo/example

You can try out a live demo of Supler. It has a moderately complex form with some subforms, custom validations, various types of inputs, and form-modifing actions.

The sources for the example are on GitHub, in the examples directory.

1.4 Form definition: Basics

You can use any object and class as the backing object for a Supler form. The fields of the class which are editable in the form must be either `var`-s, provide a Scala-style getter/setter, or immutable `val`-s in a `case class`. We recommend the latter, immutable approach.

A form consists of a list of fields belonging to a single class. Supler provides a convenience method, `form`, which captures the class of the object once, and can be used to quickly build forms. Here we are defining a form with three fields:

```
case class Person(firstName: String, lastName: String, age: Int)

val personForm = form[Person](f => List(
  f.field(_.firstName),
  f.field(_.lastName),
  f.field(_.age)
))
```

Note: An (editable) field can be used on a form only once.

The fields can be further customized. Almost always you'll want to specify the label of a field:

```
f.field(_.firstName).label("First name")
```

The label can also be a key which will be looked up in the *i18n* component on the frontend.

You can also provide a field description, which can be an explanation why you need this field to be filled, or a tip to the user on how to properly fill in a value:

```
f.field(_.firstName).label("First name").description("Your first name goes here. For example: John")
```

Note: All Supler objects (fields, forms, ...) are immutable and can be freely re-used (or shared between threads). Adding a new validator to a field, changing the label, adding fields to forms creates new field/forms instances.

1.4.1 Stand-alone fields

Fields can be created without a form, and later used to compose other forms. This may help to centralize the definition of some common fields. For example:

```
case class Person(name: String, age: Int)

val nameField = field[Person, String](_.name).label("Name")
val ageField = field[Person, Int](_.age).label("Age").validate(gt(0), le(120))

val personForm = form(_ => List(nameField, ageField))
```

1.4.2 Multiple fields in rows

If you would like to render a couple of fields in a single row, as opposed to the default horizontal rendering, you can use the `||` operator:

```
case class Person(firstName: String, lastName: String, age: Int)

val personForm = form[Person](f => List(
```

```
  f.field(_.firstName) || f.field(_.lastName),
  f.field(_.age)
))
```

the above example will render first name and last name in the first row and the age in second.

Note: The default twitter-bootstrap based frontend implementation is constrained for a maximum of 12 fields in a row. You can specify more, but the extra fields will be always wrapped in a new row.

1.5 Form definition: Supported types, type transformers

Fields of basic types (`String`, `Int`, `Long`, `Float`, `Double` and `Boolean`), `UUID` and `Date` are supported out-of-the-box, and can be directly edited in form fields.

If you have a more complex type which serializes to a basic type, you need to provide an implicit implementation of a `BasicTypeTransformer[U, S]`, where `U` is your type, and `S` is one of the basic types. For convenience, you can extend `StringTransformer[U]` etc.

In the transformer, you need to implement a method which serializes your type to a basic type, and another method which deserializes a basic type into your type, or returns a form error (conversion error).

1.5.1 Adding render hint automatically

You may wish to add render hints on some types of transformation automatically. In that case override `def renderHint: Option[RenderHint with BasicFieldCompatible]` in your transformer.

1.5.2 Example: Joda-Time DateTime transformer

The Joda-Time DateTime transformer can look like this:

```
implicit val dateTimeTransformer = new StringTransformer[DateTime] {
  override def serialize(t: DateTime) = ISODateTimeFormat.date().print(t)

  override def deserialize(u: String) = try {
    Right(ISODateTimeFormat.date().parseDateTime(u))
  } catch {
    case e: IllegalArgumentException => Left("error_custom_illegalDateFormat")
  }

  override def renderHint = Some(asDate())
}
```

1.5.3 Transforming to a json object

It is also possible to transform a value to a complex json object (not a plain string/number/boolean). To do that, you need to provide an implicit `JsonTransformer`. For example, if you have a `Point` class with two fields, a transformer which serializes it to a two-field json, and deserializes from a two-field json can take the following form:

```
implicit val pointJsonTransformer: JsonTransformer[Point] = new JsonTransformer[Point] {
  override def typeName = "point"
```

```
  override def fromJValue(jvalue: JValue) = (for {
    JObject(fields) <- jvalue
    JField("x", JInt(x)) <- fields
    JField("y", JInt(y)) <- fields
  } yield Point(x.toInt, y.toInt)).headOption

  override def toJValue(value: Point) = Some(
    JObject(JField("x", JInt(value.x)), JField("y", JInt(value.y))))
}
```

However, if the JSON representation of a field is a complex object, you will need to add support for that in the frontend as well. See *Frontend: Handling fields serialized to json objects* for more information

1.6 Form definition: Subforms

Fields can also have "complex" types and correspond to other forms. Already defined forms can be freely re-used multiple times (forms are immutable). Currently single-valued subforms, optional subforms, and ‘List‘s/‘Vector‘s of subforms are supported. For example:

```
case class Car(make: String, year: Int)
case class Person(name: String, cars: List[Car])

val carForm = ...

val personForm = form[Person](f => List(
  f.field(_.name).label("Name"),
  f.subform(_.cars, carForm).label("Cars")
))
```

By default subforms are rendered as a list. You can also render subforms as a table, see the section on *render hints*.

1.7 Form definition: Validation

Another useful customization of fields is specifying validators. There's a number of built-in validators, but you can also specify custom ones. Validators have access to the value of the field and the whole object:

```
val personForm = form[Person](f => List(
  f.field(_.firstName).label("First name")
    .validate(custom((v, e) => v.startsWith("A"), (v, e) => ErrorMessage("First name cannot start wit
  f.field(_.lastName).label("Last name"),
  f.field(_.age).label("Age").validate(ge(0), le(120))
))
```

The built-in validators include a JSON representation, and they will be checked both on the client and server side. Custom validators by default are checked only on the server, but it is possible to provide a JSON representation as well.

You can validate any object at any time using the `doValidate` method, which returns an optional list of validation errors found (the object doesn't have to come from the Supler-frontend):

```
val validationErrors: Option[FormErrors] = personForm(Person("Adam", "Smith", 18)).doValidate()
```

1.7.1 Optional fields

To validate optional fields, you can use the `ifDefined` combinator to create a validator for `Option[U]` basing on a validator for `U`; such a validator will only run the validation if the value is defined (a `Some`). For example:

```
case class Person(..., address: Option[String])

val personForm = form[Person](f => List(
  ...,
  f.field(_.address).label("Address").validate(ifDefined(maxLength(1024))
))
```

1.8 Form definition: Select fields

Very often a field can take a value from a restricted range of values. In such cases you can use select fields. You need to provide a function which, optionally depending on the backing object, returns a list of possible values.

Unlike normal fields, select fields don't require a full transformer which allows the value to be serialized to JSON and back. It is sufficient to provide a label-for-value generating function.

By default, when applying selections from JSON values are looked up by their index on the list. You can specify custom ids for select values by providing a `idForValue` function, which given a value, returns a string or number id. These ids will then be used instead of list index when rendering and serializing the form on the frontend.

1.8.1 Select one field

If only one value can be selected, the possible values function should return a `List[U]`, while the field value should be a single `U` or an `Option[U]`.

For example:

```
case class CarMake(name: String)
case class Car(make: CarMake, year: Int)

val carForm = form[Car](f => List(
  f.selectOneField(_.make)(_.name)
    .possibleValues(c => List(CarMake("Ford"), CarMake("Toyota"), CarMake("KIA")))
    .label("Make"),
  f.field(_.year).label("Year")
))
```

Here `_.name` is the function that generates `String` labels for a value.

By default select-one fields are rendered as a dropdown. They can be also rendered as radio buttons.

1.8.2 Select many field

If multiple values can be selected, the possible values function should return a `List[U]`, while the field value should be a `Set[U]` (there's no ordering):

```
case class Person(name: String, favoriteColors: Set[String])

val personForm = form[Person](f => List(
  f.field(_.name).label("Name"),
  f.selectManyField(_.favoriteColors, identity)
```

```
    .possibleValues(_ => Set("red", "green", "blue", "black"))
    .label("Favorite colors")
))
```

Note that here the label is the same as the value (`identity` is the label-for-value function).

Select-many fields are rendered as checkboxes.

1.9 Form definition: Render hints

In some cases there are a couple possible renderings of a field. In such case, you can specify a render hint, which will influence how the field is rendered. It is also possible to specify custom rendering.

For example, to render a text field as a password:

```
case class Login(username: String, password: String)

val loginForm = form[Login](f => List(
  f.field(_.username).label("Username"),
  f.field(_.password).label("Password").renderHint(asPassword())
))
```

Supported render hints:

- for subforms: `asList()` (default), `asTable()`
- for text fields: `asPassword()`, `asTextarea(rows = 10)`, `asHidden()`
- for single-select fields: `asRadio()`, `asDropdown()`

1.9.1 Custom render hints

You can also specify custom render hints which can be used as selectors for field options or for templates, customizing how fields are displayed. To create a custom render hint, you can use the `customRenderHint(name)` method. If the render hint takes additional data, you can provide any number of JSON fields:

```
f.field(_.password).label("Password").renderHint(customRenderHint("blinking", JField("interval", JInt
```

1.10 Form definition: Actions

Forms can contain buttons which invoke actions on the server side. Each action must have a unique name (fields also have names, but unlike action names, they are automatically inferred). An action name can only contain letters, digits and _ (no spaces or other characters which would form an invalid JSON object key).

In its simplest form, an action can modify the object that is backing the form, and needs to return an `ActionResult`:

```
case class Person(name: String)

val personForm = form[Person](f => List(
  f.field(_.name).label("Name"),
  f.action("duplicateName")(p => ActionResult(p.copy(name = s"${p.name} ${p.name}"))
    .label("Duplicate name")
))
```

Actions can result not only in modified objects, but also return some custom data (JSON) to the client. You can either return both an object and custom data using `ActionResult(obj, Some(jvalue))`, or only custom data using `ActionResult.custom(jvalue)`. Depending on the variant, when the JSON is generated, the custom data will be placed next to the form data, or will replace the whole generated JSON.

To implement some operations on subforms, such as removing a subform element, or moving the elements around, it is necessary to have access to the parent object. This is possible by using ‘‘parentAction‘‘s. The subform is in such case parametrised by the action (so it can be reused in different contexts), which is provided in the parent form:

```
case class Address(street: String)
case class Person(name: String, addresses: List[Address]) {
  def removeAddress(a: Address) = this.copy(addresses = this.addresses diff List(a))
}

def addressForm(removeAction: Address => ActionResult[Address]) = form[Address](f => List(
  f.field(_.street).label("Street"),
  f.action("remove")(removeAction).label("Remove")
))

val personForm = form[Person](f => List(
  f.field(_.name).label("Name"),
  f.subform(_.addresses, addressForm(
    f.parentAction((person, index, address) => ActionResult(person.removeAddress(address)))))
    .label("Addresses")
))
```

To enable sending form content automatically when an action is invoked, see the section on *frontend refreshes*).

1.10.1 Before-action validation

Note: Before an action is run, the form can be validated. By default, no validations are run before an action is invoked. You can customize that behavior using the `.validate...()` methods on an action field.

You can validate either the entire form, the current subform or run no validations at all. For example, when implementing a “save form” action, you will most probably want to validate all fields in the form:

```
val personForm = form[Person](f => List(
  f.field(_.name).label("Name"),
  f.action("save")(p => ActionResult(persist(p))).label("Save").validateAll()
))
```

1.11 Form definition: Label (static) fields

Fields can also be non-editable and display static content - a label. Note that the value of such fields will **not** be included when the form is serialized on the frontend, and sent back to the server:

```
case class Person(name: String, registrationId: String)

val personForm = form[Person](f => List(
  f.field(_.name).label("Name"),
  f.staticField(_.registrationId).label("Registration id")
))
```

Static fields have random names by default. If you’d like to customize the field order, assign a name to the static field using the `name` method.

1.12 Form definition: Conditional fields

Form fields and subforms can be included and enabled conditionally, basing on the backing object's state. The conditions can be defined using:

- `f.field(_.likesChocolate).includeIf(person => person.hasNoAllergies)` (included or not)
- `f.field(_.likesChocolate).enabledIf(person => person.hasNoAllergies)` (enabled / disabled)

If a field is not included (the test returns `false`), the field won't be included in the JSON form representation sent to the frontend. Hence, the field won't be visible at all.

If a field is disabled, it will be disabled in the frontend (usually grayed out), but still present in the JSON form representation.

Both not included and disabled fields are not taken into account when applying values to a backing object from a JSON object.

However, fields are always validated (even if not visible in the frontend or disabled). This is because validation checks the object as a whole, and even when not editable, all field values should be valid. A failing validation of a hidden field is most probably an error in the form definition. If you need to conditionally run validations, probably the best choice is using optional fields (or optional subforms, which can be used to conditionally validate a group of fields); the "conditionality" will then be also reflected in the model. Alternatively, you can use a custom validator.

1.13 Backend: form with object

By applying an object to a form definition, we get a `FormWithObject[T]` instance which contains a number of operations, allowing to generate a JSON form description, apply new values to the object and validate the current state.

1.13.1 Applying values and validating

After receiving a JSON representing an updated state, the form can be used to apply the values to an object:

```
personForm(person).applyValuesFromJSON(receivedJson)
```

This can then be chained with validation, or validation can be run on any object:

```
personForm(person).doValidate()

// chained:
personForm(person).applyValuesFromJSON(receivedJson).doValidate()
```

The resulting type of each method is a `FormWithObject[Person]`, which contains potential conversion/validation errors and the current state of the object.

When validating, there is a special mode which runs the validations only for fields with filled-in values, not to show the user validation errors for fields which haven't been yet edited at all: `doValidate(ValidateFilled)`. This is useful when validating partially-filled forms, and is also what `process` (described below) does when no action is invoked.

1.13.2 Creating new objects

If you don't have a starting object, you can create an "empty" one and apply values to it:

```
personForm.withNewEmpty.applyValuesFromJSON(receivedJson)
```

The `withNewEmpty` method returns a `FormWithObject[Person]`, just as applying an existing object, so you can use it in the same way.

By default, all fields in the object will have "empty" values (depending on their type); most probably, validation for such an empty object will fail. You can also customize how to create new empty objects, using the `Form.useNewEmpty(newCreateEmpty: => T)` method.

1.13.3 Serializing a form to JSON

At any stage it is possible to convert the current state to JSON. In most cases the JSON will contain the form representation. If an action was run which results in custom JSON only, the result will contain only that data.

The JSON form representation contains both the form structure and the form values. It is a custom format, however it's very easy to understand, and self-explanatory; the fields in the JSON correspond closely to the DSL-based definition. In case there were validation or conversion errors, they will be included as well.

To generate the JSON representation, simply call the `generateJSON` method:

```
val personFormJson = personForm(person).generateJSON
```

The resulting JSON can be then sent to the client. Supler uses the Scala-standard json4s to generate the JSON.

1.13.4 Processing JSON

The `FormWithObject.process(JValue)` method was already described in the *introduction*, so just a short recap; process:

- converts and applies values from the given json to the backing object
- runs validations (if no action, only for filled fields, otherwise for the scope specified by the action)
- if there are no errors, runs the actions (if any)

This represents the most common flow when working with Supler. The result of `process` is a `SuplerData` instance, which can either be a `FormWithObject` or, if a custom-data-only action was run, `CustomDataOnly`.

1.14 Backend: form with meta

In some cases you will find it useful to send some payload with the form that you should be able to extract before processing the form with object.

One good example is sending an entity id of the object backing the form, so that on each POST you will be able to pull it from the database and apply changes that came from the frontend.

1.14.1 Add meta on form creation

When creating the form use the withMeta syntax:

```
case class Person(id: String, name: String)

val personForm = form[Person](f => List(
  f.field(_.name).label("Name")
))
```

```
personForm(person).withMeta("id", person.id).generateJSON
```

which will add the needed payload to the form.

1.14.2 Extract the meta on form posts

The meta can be later extracted on POST calls with:

```
val id = personForm.getMeta(jsonBody)("id")
val person = personDao.lookup(id)
personForm(person).process(jsonBody).generateJSON
```

where jsonBody is the serialized object that comes from the frontend.

1.15 Frontend: Rendering the form

1.15.1 Form container

At the minimum, you need a designated container on your page, where the form will be rendered, and when the form JSON is available, create a new `Supler.Form`:

```
<div>
  <div id="form-container"></div>
  <a href="#" class="btn btn-primary btn-lg" id="submit" role="button">Submit</a>
  <p id="feedback"></p>
</div>
```

```
var formContainer = document.getElementById('form-container');
var form = new Supler.Form(formContainer, {});
form.render(formJson); // formJson is received from the server
```

If the JSON received from the server contains validation errors, they will be displayed as well.

1.15.2 Field order

You may choose to change the order of fields that comes from the backend. To do so, override `field_order` form option.

```
new Supler.Form(container, {
  field_order: [
      ['firstName', 'lastName'],
      ['address'],
      ['street', 'streetNo', 'apptNo'],
      ['postcode', 'city', 'country']
  ]
});
```

The `field_order` field should be a two-dimensional string array with field names that represents form rows.

Note that if you have fields which have random names (static fields), you will have to name them to be able to reference them in the field order.

1.16 Frontend: Client-side validation

To perform and display client side validation, use the `Form.validate()` method. It will return `true` if there are any validation errors.

Any existing errors will be cleared upon next invocation of `validate()`.

1.17 Frontend: Serializing the form

To read the value of a form as a JSON object, simply use the `Supler.Form.getValue()` method. The resulting JSON can be sent to the server for processing.

The resulting JSON is what you might expect, mirroring the form's structure through objects, JSON arrays, nested objects and primitive types.

In fact, to apply a JSON to an object on the server-side you don't need to use Supler-frontend. Because there's nothing special about the format, it is easy to generate such a JSON yourself.

1.18 Frontend: Customizing the rendering process

The rendering process is fully customizable. By default, Bootstrap-based HTML is rendered, but this can be changed either by providing HTML templates, or by overriding any of the rendering functions using the options.

1.18.1 Defining render hints on the frontend

Render hints can be specified per-field and influence how a field is rendered. All of the render hints supported by default can be defined on the *backend*, but it is also possible to define them on the frontend.

This can be done through the the `field_options` option passed when creating a form:

```
new Supler.Form(container, {
  field_options: {
    'secretField': {
      'render_hint': 'password'
    },
    'friends[].bio': {
      'render_hint': {
        'name': 'textarea',
        'rows': 10,
        'cols': 20
      }
    }
  }
});
```

A render hint can be just a name (string), or an object with a `name` property and additional parameters (like the textarea example).

1.18.2 Customizing via HTML templates

The generated HTML can be customized by providing templates, which will be used during the rendering process. By default Supler looks for templates nested inside the element that will contain the form. For example:

```
<div id="form-container">
  <div supler:fieldTemplate supler:fieldPath="lastName">
    <div class="formGroup">
      <i>{{suplerLabel}} <span style="font-size: xx-small">(extra information)</span></i>
      {{suplerInput}}
      {{suplerValidation}}
    </div>
  </div>
</div>
```

If the templates are defined somewhere else, you can provide additional ids of the elements from which templates should be read by defining the `field_templates` option:

```
new Supler.Form(container, {
  field_templates: [ 'idOfElementWithTemplates1', 'idOfElementWithTemplates2' ]
});
```

The templates are stacked top-to-bottom, that is the templates that are defined higher will take precedence, if multiple templates match a given field.

Matching templates to fields

The templates are applied to all matching fields. The matchers should be specified as attributes of the template. Currently the following matchers are allowed:

- `supler:fieldPath="..."` where field path can be e.g. `cars.model.name`
- `supler:fieldType="..."` where type can be e.g. `string`, `integer`, `double`, `static` etc.
- `supler:fieldRenderHint="..."` where the render hint can be e.g. `textarea`, `password`, `radio` etc.

That way templates for a specific field or field type can be specified.

Types of templates

- re-define the template for rendering fields

```
<div id="form-container">
  <div supler:fieldTemplate>
    // html with placeholders:
    // {{suplerLabel}}
    // {{suplerDescription}}
    // {{suplerInput}}
    // {{suplerValidation}}
  </div>
</div>
```

- re-define how labels are rendered

```
<div id="form-container">
  <div supler:fieldLabelTemplate>
    // html with placeholders:
    // {{suplerLabelForId}}
    // {{suplerLabelText}}
  </div>
</div>
```

- re-define how descriptions are rendered

```
<div id="form-container">
  <div supler:fieldDescriptionTemplate>
    // html with placeholders:
    // {{suplerDescriptionText}}
  </div>
</div>
```

- re-define how validations are rendered

```
<div id="form-container">
  <div supler:fieldValidationTemplate>
    // html with placeholders:
    // {{suplerValidationId}}
  </div>
</div>
```

- re-define how a field's input without possible values is rendered

```
<div id="form-container">
  <div supler:fieldInputTemplate>
    // html with placeholders:
    // {{suplerFieldInputAttrs}}
    // {{suplerFieldInputValue}}
  </div>
</div>
```

This should always be combined with a filter to make sense. The attributes will contain normal attributes such as `id`, `name`, as well as supler-specific meta-data. If `{{suplerFieldInputValue}}` is used, the attributes won't include the field value (useful e.g. for textarea fields). Otherwise the attributes will contain the value mapping.

- re-define how a field's input with possible values is rendered

```
<div id="form-container">
  <div supler:fieldInputTemplate supler:singleInput="true|false" supler:selectedAttrName="selected" s
    // html with placeholders:
    // {{suplerFieldInputContainerAttrs}}
    // must contain an element with the "supler:possibleValueTemplate" attribute;
    // that element will be repeated for each possible value. Placeholders:
    // {{suplerFieldInputAttrs}}, {{suplerFieldInputValue}}, {{suplerFieldInputLabel}}
  </div>
</div>
```

To properly render a field with possible values, Supler needs to know if the element is rendered as a single input (e.g. drop-down) or multiple inputs (e.g. radio/checkboxes).

Also, if an element is already selected, it must have an additional attribute, which will be added to the possible value template. The attribute name & value are specified using `supler:selectedAttrName` and `supler:selectedAttrValue`.

Not yet implemented

- re-define how a field overall is given (without separating into label/input/validation)

```
<div id="form-container">
  <div supler:fieldFlatTemplate>
    // html with placeholders:
    // {{suplerFieldInputAttrs}}
```

```
    // {{suplerFieldLabelForId}}
    // {{suplerFieldLabelText}}
    // {{suplerFieldDescriptionText}}
    // {{suplerFieldValidationId}}
  </div>
</div>
```

- re-define how a subform is rendered

```
<div id="form-container">
  <div supler:subformDecorationTemplate>
    // html with placeholders:
    // {{suplerSubformLabel}}
    // {{suplerSubform}}
    // {{suplerSubformContainerAttrs}}
  </div>
</div>
```

- re-define how a subform element is rendered (as-list rendering)

```
<div id="form-container">
  <div supler:subformListElementTemplate>
    // html with placeholders:
    // {{suplerSubformElement}}
    // {{suplerSubformElementContainerAttrs}}
  </div>
</div>
```

- re-define how a subform element is rendered (as-table rendering)

```
<div id="form-container">
  <div supler:subformTableTemplate>
    // html with placeholders:
    // {{suplerSubformTableHeaders}}
    // {{suplerSubformTableCells}}
  </div>
</div>
```

The table headers are a series of `<tr><th>` tags. The table cells are a series of `<tr><td></td><td></td>..></tr>...` tags.

- re-define the order of fields

```
<div id="form-container" supler:fieldOrder="x, y, z">
</div>
```

1.18.3 Customizing via local javascript options

Rendering can also be customized by providing customizations using javascript instead of HTML templates. You can override any of the methods available on `RenderOptions` (see below for a complete list) using field options:

```
new Supler.Form(container, {
  field_options: {
    'bio': {
      'render_options': {
        renderLabel: function(forId, label) { return '<div>some html</div>'; }
      }
    }
```

```
  }
});
```

It is also possible to match using render hints, instead of field names/paths. You need to prefix the field option name with `render_hint:`. For example, to provide custom javascript rendering options for all fields with render hint `date`:

```
new Supler.Form(container, {
  field_options: {
    'render_hint:date': {
      'render_options': {
        renderLabel: function(forId, label) { return '<div>this is a date</div>'; }
      }
    }
  }
});
```

1.18.4 Customizing via global javascript options

To override how particular types of form elements are rendered globally, simply provide a method in the `render_options` option passed to `Supler.Form`; you can even provide a whole alternative implementation of the `RenderOptions` interface:

```
var formContainer = document.getElementById('form-container');
var form = new Supler.Form(formContainer, {
  render_options: {
    renderStringField: function(label, id, validationId, name, value, options, compact) {
      return someHtml;
    }
  }
});
form.render(formJson); // formJson is received from the server
```

Methods available for overriding:

```
// basic types
renderTextField: (fieldData: FieldData, options: any, compact: boolean): string
renderHiddenField: (fieldData: FieldData, options: any, compact: boolean): string
renderTextareaField: (fieldData: FieldData, options: any, compact: boolean): string
renderMultiChoiceCheckboxField: (fieldData: FieldData, possibleValues: SelectValue[], options: any, c
renderMultiChoiceSelectField: (fieldData: FieldData, possibleValues: SelectValue[], options: any, con
renderSingleChoiceRadioField: (fieldData: FieldData, possibleValues: SelectValue[], options: any, con
renderSingleChoiceSelectField: (fieldData: FieldData, possibleValues: SelectValue[], options: any, co
renderActionField: (fieldData: FieldData, options: any, compact: boolean): string

// templates
// [label] [input] [validation]
renderField: (input: string, fieldData: FieldData, compact: boolean) => string
renderLabel: (forId: string, label: string) => string
renderDescription: (description:string) => string
renderValidation: (validationId: string) => string

renderRow: (fields: string) => string

renderForm: (rows: string) => string

renderStaticField: (label: string, id: string, validationId: string, value: any, compact: boolean) =>
```

```
renderStaticText: (text: string) => string

renderSubformDecoration: (subform: string, label: string, id: string, name: string) => string
renderSubformListElement: (subformElement: string, options: any) => string;
renderSubformTable: (tableHeaders: string[], cells: string[][], elementOptions: any) => string;

// html form elements
renderHtmlInput: (inputType: string, value: any, options: any) => string
renderHtmlSelect: (value: number, possibleValues: SelectValue[], options: any) => string
renderHtmlRadios: (value: any, possibleValues: SelectValue[], options: any) => string
renderHtmlCheckboxes: (value: any, possibleValues: SelectValue[], options: any) => string
renderHtmlTextarea: (value: string, options: any) => string

// misc
additionalFieldOptions: () => any
inputTypeFor: (fieldData:FieldData) => string
```

1.19 Frontend: i18n

Both the labels and the conversion/validation errors may be i18n keys (any place on the backend of frontend that takes a user-visible string, can be a i18n key instead). Some default keys are provided for the standard validators, but custom ones can be provided as well simply by specifying them as members of the `i18n` option to `Supler.Form`:

```
var formContainer = document.getElementById('form-container');
var form =  = new Supler.Form(formContainer, {
  i18n: {
    error_custom_lastNameLongerThanFirstName: "Last name must be longer than first name!",
    error_custom_illegalDateFormat: function(detail) { return "Illegal date format: " + detail; }
  }
});
form.render(formJson); // formJson is received from the server
```

The values can be either strings, or functions which format the message using the error message's arguments.

1.20 Frontend: Updating the form basing on server-side form changes

The form can be automatically updated after each field edit (value change) and when actions are performed. To do that, two things are necessary. First, a `send_form_function` option must be specified. This should be a javascript function, accepting form representation (as a JS object) and callbacks for handling response and errors, to be called when the backend responds with an updated form representation or if the request fails. For example, when using JQuery, this can be:

```
function sendForm(formValue, renderResponseFn, sendErrorFn, isAction, triggeringElement) {
    $.ajax({
        url: '/refresh_form.json',
        type: 'POST',
        data: JSON.stringify(formValue),
        dataType: 'json',
        contentType: 'application/json; charset=utf-8',
        success: renderResponseFn,
        error: sendErrorFn
    });
```

```
}

var form = new Supler.Form(formContainer, {
  send_form_function: sendForm,
  // other options
});
```

Secondly, we need to provide a server-side endpoint which will refresh the form with the given values, validate and generate back the response (the server can use the *process* method, or any other way of processing the data).

Concurrent sends are handled as well. Only the results of the last send triggered by value changes will be taken into account. Only one action can be in progress at a time (hence errors must be reported using `sendErrorFn` to allow subsequent actions to execute after a failed one). It could be a good idea to block the UI while an action is executing, so that no form changes are made during action execution (which would be lost). The `isAction` flag can be used to determine, if the callback is triggered by an action (there is usually no need to block the UI for value-change refreshes).

1.21 Frontend: Adding custom behavior to the form

By setting the `after_render_function` option to a no-argument function, it is possible to get notified after a form is rendered (or refreshed), and customize the form or add some custom dynamic behavior. An example of such customization in *the live-demo example*, is using a stylized date picker plugin for date fields.

1.22 Frontend: Handling custom data

Actions can result in custom data being returned by the server. Custom data can come either together with a form, or without. There are two ways to handle custom data. First, you can specify the `custom_data_handler` option, which should be a function accepting the data object. The function will be invoked after rendering the form with `render` (or calling the `renderResponseFn`):

```
function handleData(data) {
    $('#messages').html('Server response: ' + data);
}

var form = new Supler.Form(formContainer, {
  custom_data_handler: handleData,
  // other options
});
```

The second way is to get the custom data by hand, using `Form.getCustomData(json)` (where `json` is what you receive from the server), and if the result is not null, handling it as desired.

1.23 Frontend: Options reference

Here's a summary of all options that can be used when defining a Supler form.

```
new Supler.Form(container, {
  send_form_function: doAjax, // [1]
  i18n: { // [2]
    error_custom_someDescription: 'You cannot do that!',
    error_custom_complex: function(parameter) { return parameter + ' is a bad choice'; }
  },
  field_options: {
```

```
    'secretField': {
      render_hint: 'password' // [3]
    },
    'friends[].bio': {
      render_options: {
        renderLabel: function(forId, label) { return '<div>some html</div>'; } // [4]
      }
      read_value: function(element) { return ... } // [5]
    },
    'render_hint:radio': {

    }
  },
  after_render_function: enrichForm, // [6]
  custom_data_handler: displayCustomData, // [7]
  validators: {
    good_value: function(json) { return (fieldValue) => { return "error"; } }
  },
  render_options: new Bootstrap3RenderOptions(), // or any subset of methods from RenderOptions
  field_templates: [ 'idOfElementWithTemplates1', 'idOfElementWithTemplates2' ], // [8]
  field_order: [
      ['firstName', 'lastName'],
      ['address']
  ] // [9]
});
```

When specifying field options and dealing with lists of subforms, options for nested fields can be defined using the `subformField[].fieldName` syntax (`[]` means every subform in subforms list). If you want to specify options for single subforms and fields, you can use indexes such as `subformField[2].fieldName`.

Options details:

- [1] *Frontend: Updating the form basing on server-side form changes*
- [2] *Frontend: i18n*
- [3] *Defining render hints on the frontend*
- [4] *Customizing via local javascript options*
- [5] *Frontend: Handling fields serialized to json objects*
- [6] *Frontend: Adding custom behavior to the form*
- [7] *Frontend: Handling custom data*
- [8] *Customizing via HTML templates*
- [9] *Field order*

1.24 Frontend: Handling fields serialized to json objects

If a field is serialized to a JSON object (not a basic type, such as a string, number or boolean), we will need to add custom code to display the field and read its value. To see how to (de)serialize fields to JSON objects on the backend, see *the documentation on type transformers*.

For example, if the field is serialized to an object with two fields: `x` and `y` (a point), we need to provide a method which renders two inputs instead of one, and which reads value from those two inputs and creates an object.

We can do both of these things by providing field options. If the name of the field is `pointField`:

```
var sf = new Supler.Form(container, {
  field_options: {
    pointField: {
      render_options: {
        renderHtmlInput: function (inputType, value, options) {
          return Supler.HtmlUtil.renderTag('span', options,
            Supler.HtmlUtil.renderTag('input',
              { class: 'x-coord', type: 'number', value: value.x }) +
            Supler.HtmlUtil.renderTag('input',
              { class: 'y-coord', type: 'number', value: value.y })
          );
        }
      },
      read_value: function(element) {
        return {
          x: parseInt($('.x-coord', element).val()),
          y: parseInt($('.y-coord', element).val())
        }
      }
    }
  }
});
```

To properly display the field, we override the `renderHtmlInput` method of the render options that will be used for rendering the field. In the method, we create a container element which has all the supler-specific attributes (passed as `options`). These attributes will be used to identify elements from which field values can be later read.

We are using the `Supler.HtmlUtil.renderTag` helper method, which simply renders a tak with the given attributes and body. The body are two inputs: one for the `x`, and one for the `y` coordinate.

Secondly, we provide a custom field-value reading method, by specifying the `read_value` field option. This option takes an element, from which the value should be read (here, this will be the rendered `span`). The return value should be the json object, which will be then passed to the backend.

1.25 Supler-generated JSON

Supler generates two kinds of JSON objects:

- the backend: form description, with field values, validation errors
- the frontend: serialized form with field values

Both are quite easy to understand without knowing much about how Supler works. For example, the JSON form description corresponding to the form from *the live demo* is:

```
{
  "supler_meta": {
  },
  "is_supler_form": true,
  "main_form": {
    "fields": [
      {
        "name": "firstName",
        "enabled": true,
        "label": "label_person_firstname",
        "type": "string",
        "validate": {
          "required": true
```

```
      },
      "path": "firstName",
      "value": "Adam",
      "empty_value": ""
    },
    {
      "name": "lastName",
      "enabled": true,
      "label": "label_person_lastname",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "lastName",
      "value": "",
      "empty_value": ""
    },
    {
      "name": "age",
      "enabled": true,
      "label": "Age",
      "type": "integer",
      "validate": {
        "required": true
      },
      "path": "age",
      "value": 10,
      "empty_value": 0
    },
    {
      "name": "birthday",
      "enabled": true,
      "label": "Birthday",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "birthday",
      "value": "2015-03-09"
    },
    {
      "name": "likesBroccoli",
      "enabled": true,
      "label": "Likes broccoli",
      "type": "boolean",
      "validate": {
        "required": true
      },
      "path": "likesBroccoli",
      "value": false
    },
    {
      "name": "address1",
      "enabled": true,
      "label": "Address 1",
      "type": "string",
      "validate": {
        "required": false
```

```
      },
      "path": "address1"
    },
    {
      "name": "address2",
      "enabled": true,
      "label": "Address 2",
      "type": "string",
      "validate": {
        "required": false
      },
      "path": "address2"
    },
    {
      "name": "favoriteColors",
      "enabled": true,
      "label": "Favorite colors",
      "type": "select",
      "validate": {
        "required": false
      },
      "path": "favoriteColors",
      "value": [
        "0",
        "2"
      ],
      "possible_values": [
        {
          "id": "0",
          "label": "red"
        },
        {
          "id": "1",
          "label": "green"
        },
        {
          "id": "2",
          "label": "blue"
        },
        {
          "id": "3",
          "label": "magenta"
        }
      ],
      "multiple": true
    },
    {
      "name": "gender",
      "enabled": true,
      "label": "Gender",
      "type": "select",
      "validate": {
        "required": true
      },
      "path": "gender",
      "value": null,
      "empty_value": null,
      "render_hint": {
```

```
          "name": "radio"
        },
        "possible_values": [
          {
            "id": "0",
            "label": "Male"
          },
          {
            "id": "1",
            "label": "Female"
          }
        ]
      },
      {
        "name": "secret",
        "enabled": true,
        "label": "Secret",
        "type": "string",
        "validate": {
          "required": false
        },
        "path": "secret",
        "render_hint": {
          "name": "password"
        }
      },
      {
        "name": "bio",
        "enabled": true,
        "label": "Biography",
        "type": "string",
        "validate": {
          "required": false
        },
        "path": "bio",
        "render_hint": {
          "name": "textarea",
          "rows": 6
        }
      },
      {
        "name": "cars",
        "enabled": true,
        "type": "subform",
        "render_hint": {
          "name": "list"
        },
        "multiple": true,
        "label": "Cars",
        "path": "cars",
        "value": [
          {
            "fields": [
              {
                "name": "make",
                "enabled": true,
                "label": "Make",
                "type": "select",
```

```
      "validate": {
        "required": true
      },
      "path": "cars[0].make",
      "value": "0",
      "empty_value": null,
      "possible_values": [
        {
          "id": "0",
          "label": "Ford"
        },
        {
          "id": "1",
          "label": "Toyota"
        },
        {
          "id": "2",
          "label": "KIA"
        },
        {
          "id": "3",
          "label": "Lada"
        }
      ]
    },
    {
      "name": "model",
      "enabled": true,
      "label": "Model",
      "type": "select",
      "validate": {
        "required": true
      },
      "path": "cars[0].model",
      "value": "1",
      "empty_value": null,
      "possible_values": [
        {
          "id": "0",
          "label": "Ka"
        },
        {
          "id": "1",
          "label": "Focus"
        },
        {
          "id": "2",
          "label": "Mondeo"
        },
        {
          "id": "3",
          "label": "Transit"
        }
      ]
    },
    {
      "name": "year",
      "enabled": true,
```

```
            "label": "Year",
            "type": "integer",
            "validate": {
              "required": true,
              "gt": 1900.0
            },
            "path": "cars[0].year",
            "value": 1990,
            "empty_value": 0
          },
          {
            "name": "delete",
            "enabled": true,
            "label": "Delete",
            "type": "action",
            "path": "cars[0].delete",
            "validation_scope": {
              "name": "none"
            }
          }
        ],
        "fieldOrder": [
          [
            "make",
            "model"
          ],
          [
            "year"
          ],
          [
            "delete"
          ]
        ]
      },
      {
        "fields": [
          {
            "name": "make",
            "enabled": true,
            "label": "Make",
            "type": "select",
            "validate": {
              "required": true
            },
            "path": "cars[1].make",
            "value": "1",
            "empty_value": null,
            "possible_values": [
              {
                "id": "0",
                "label": "Ford"
              },
              {
                "id": "1",
                "label": "Toyota"
              },
              {
                "id": "2",
```

```
            "label": "KIA"
          },
          {
            "id": "3",
            "label": "Lada"
          }
        ]
      },
      {
        "name": "model",
        "enabled": true,
        "label": "Model",
        "type": "select",
        "validate": {
          "required": true
        },
        "path": "cars[1].model",
        "value": "5",
        "empty_value": null,
        "possible_values": [
          {
            "id": "0",
            "label": "Aygo"
          },
          {
            "id": "1",
            "label": "Yaris"
          },
          {
            "id": "2",
            "label": "Corolla"
          },
          {
            "id": "3",
            "label": "Auris"
          },
          {
            "id": "4",
            "label": "Verso"
          },
          {
            "id": "5",
            "label": "Avensis"
          },
          {
            "id": "6",
            "label": "Rav4"
          }
        ]
      },
      {
        "name": "year",
        "enabled": true,
        "label": "Year",
        "type": "integer",
        "validate": {
          "required": true,
          "gt": 1900.0
```

```
            },
            "path": "cars[1].year",
            "value": 2004,
            "empty_value": 0
          },
          {
            "name": "delete",
            "enabled": true,
            "label": "Delete",
            "type": "action",
            "path": "cars[1].delete",
            "validation_scope": {
              "name": "none"
            }
          }
        ],
        "fieldOrder": [
          [
            "make",
            "model"
          ],
          [
            "year"
          ],
          [
            "delete"
          ]
        ]
      }
    ]
  },
  {
    "name": "addcar",
    "enabled": true,
    "label": "Add car",
    "type": "action",
    "path": "addcar",
    "validation_scope": {
      "name": "none"
    }
  },
  {
    "name": "legoSets",
    "enabled": true,
    "type": "subform",
    "render_hint": {
      "name": "table"
    },
    "multiple": true,
    "label": "Lego sets",
    "path": "legoSets",
    "value": [
      {
        "fields": [
          {
            "name": "name",
            "enabled": true,
            "label": "label_lego_name",
```

```
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "legoSets[0].name",
      "value": "Motorcycle",
      "empty_value": ""
    },
    {
      "name": "theme",
      "enabled": true,
      "label": "label_lego_theme",
      "type": "select",
      "validate": {
        "required": true
      },
      "path": "legoSets[0].theme",
      "value": "1",
      "empty_value": null,
      "possible_values": [
        {
          "id": "0",
          "label": "City"
        },
        {
          "id": "1",
          "label": "Technic"
        },
        {
          "id": "2",
          "label": "Duplo"
        },
        {
          "id": "3",
          "label": "Space"
        },
        {
          "id": "4",
          "label": "Friends"
        },
        {
          "id": "5",
          "label": "Universal"
        }
      ]
    },
    {
      "name": "number",
      "enabled": true,
      "label": "label_lego_setnumber",
      "type": "integer",
      "validate": {
        "required": true,
        "lt": 100000.0
      },
      "path": "legoSets[0].number",
      "value": 1924,
      "empty_value": 0
```

```
        },
        {
          "name": "age",
          "enabled": true,
          "label": "label_lego_age",
          "type": "integer",
          "validate": {
            "required": true,
            "ge": 0.0,
            "le": 50.0
          },
          "path": "legoSets[0].age",
          "value": 31,
          "empty_value": 0
        },
        {
          "name": "delete",
          "enabled": true,
          "label": "Delete",
          "type": "action",
          "path": "legoSets[0].delete",
          "validation_scope": {
            "name": "none"
          }
        }
      ],
      "fieldOrder": [
        [
          "name"
        ],
        [
          "theme"
        ],
        [
          "number"
        ],
        [
          "age"
        ],
        [
          "delete"
        ]
      ]
    },
    {
      "fields": [
        {
          "name": "name",
          "enabled": true,
          "label": "label_lego_name",
          "type": "string",
          "validate": {
            "required": true
          },
          "path": "legoSets[1].name",
          "value": "Arctic Supply Plane",
          "empty_value": ""
        },
```

```
                {
                  "name": "theme",
                  "enabled": true,
                  "label": "label_lego_theme",
                  "type": "select",
                  "validate": {
                    "required": true
                  },
                  "path": "legoSets[1].theme",
                  "value": "0",
                  "empty_value": null,
                  "possible_values": [
                    {
                      "id": "0",
                      "label": "City"
                    },
                    {
                      "id": "1",
                      "label": "Technic"
                    },
                    {
                      "id": "2",
                      "label": "Duplo"
                    },
                    {
                      "id": "3",
                      "label": "Space"
                    },
                    {
                      "id": "4",
                      "label": "Friends"
                    },
                    {
                      "id": "5",
                      "label": "Universal"
                    }
                  ]
                },
                {
                  "name": "number",
                  "enabled": true,
                  "label": "label_lego_setnumber",
                  "type": "integer",
                  "validate": {
                    "required": true,
                    "lt": 100000.0
                  },
                  "path": "legoSets[1].number",
                  "value": 60064,
                  "empty_value": 0
                },
                {
                  "name": "age",
                  "enabled": true,
                  "label": "label_lego_age",
                  "type": "integer",
                  "validate": {
                    "required": true,
```

```
                "ge": 0.0,
                "le": 50.0
              },
              "path": "legoSets[1].age",
              "value": 1,
              "empty_value": 0
            },
            {
              "name": "delete",
              "enabled": true,
              "label": "Delete",
              "type": "action",
              "path": "legoSets[1].delete",
              "validation_scope": {
                "name": "none"
              }
            }
          ],
          "fieldOrder": [
            [
              "name"
            ],
            [
              "theme"
            ],
            [
              "number"
            ],
            [
              "age"
            ],
            [
              "delete"
            ]
          ]
        },
        {
          "fields": [
            {
              "name": "name",
              "enabled": true,
              "label": "label_lego_name",
              "type": "string",
              "validate": {
                "required": true
              },
              "path": "legoSets[2].name",
              "value": "Princess and Horse",
              "empty_value": ""
            },
            {
              "name": "theme",
              "enabled": true,
              "label": "label_lego_theme",
              "type": "select",
              "validate": {
                "required": true
              },
```

```
      "path": "legoSets[2].theme",
      "value": "2",
      "empty_value": null,
      "possible_values": [
        {
          "id": "0",
          "label": "City"
        },
        {
          "id": "1",
          "label": "Technic"
        },
        {
          "id": "2",
          "label": "Duplo"
        },
        {
          "id": "3",
          "label": "Space"
        },
        {
          "id": "4",
          "label": "Friends"
        },
        {
          "id": "5",
          "label": "Universal"
        }
      ]
    },
    {
      "name": "number",
      "enabled": true,
      "label": "label_lego_setnumber",
      "type": "integer",
      "validate": {
        "required": true,
        "lt": 100000.0
      },
      "path": "legoSets[2].number",
      "value": 4825,
      "empty_value": 0
    },
    {
      "name": "age",
      "enabled": true,
      "label": "label_lego_age",
      "type": "integer",
      "validate": {
        "required": true,
        "ge": 0.0,
        "le": 50.0
      },
      "path": "legoSets[2].age",
      "value": 7,
      "empty_value": 0
    },
    {
```

```
                "name": "delete",
                "enabled": true,
                "label": "Delete",
                "type": "action",
                "path": "legoSets[2].delete",
                "validation_scope": {
                  "name": "none"
                }
              }
            ],
            "fieldOrder": [
              [
                "name"
              ],
              [
                "theme"
              ],
              [
                "number"
              ],
              [
                "age"
              ],
              [
                "delete"
              ]
            ]
          }
        ]
      },
      {
        "name": "addlegoset",
        "enabled": true,
        "label": "Add lego set",
        "type": "action",
        "path": "addlegoset",
        "validation_scope": {
          "name": "none"
        }
      },
      {
        "name": "_supler_static_-1345397749",
        "enabled": true,
        "label": "Registration date",
        "type": "static",
        "validate": {
        },
        "path": "_supler_static_-1345397749",
        "value": {
          "params": [
          ],
          "key": "2012-02-19"
        }
      },
      {
        "name": "id",
        "enabled": true,
        "label": "",
```

```
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "id",
      "value": "d606796a-9f91-42ad-8d51-1e3dd203027c",
      "empty_value": "",
      "render_hint": {
        "name": "hidden"
      }
    },
    {
      "name": "a1",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a1",
      "value": "a",
      "empty_value": ""
    },
    {
      "name": "a2",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a2",
      "value": "b",
      "empty_value": ""
    },
    {
      "name": "a3",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a3",
      "value": "c",
      "empty_value": ""
    },
    {
      "name": "a4",
      "enabled": true,
      "label": "4th field",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a4",
      "value": "d",
      "empty_value": ""
```

```
    },
    {
      "name": "a5",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a5",
      "value": "e",
      "empty_value": ""
    },
    {
      "name": "a6",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a6",
      "value": "f",
      "empty_value": ""
    },
    {
      "name": "a7",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a7",
      "value": "g",
      "empty_value": ""
    },
    {
      "name": "a8",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a8",
      "value": "h",
      "empty_value": ""
    },
    {
      "name": "a9",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a9",
```

```
      "value": "i",
      "empty_value": ""
    },
    {
      "name": "a10",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a10",
      "value": "j",
      "empty_value": ""
    },
    {
      "name": "a11",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a11",
      "value": "k",
      "empty_value": ""
    },
    {
      "name": "a12",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a12",
      "value": "l",
      "empty_value": ""
    },
    {
      "name": "a13",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a13",
      "value": "m",
      "empty_value": ""
    },
    {
      "name": "a14",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
```

```
      },
      "path": "a14",
      "value": "n",
      "empty_value": ""
    },
    {
      "name": "a15",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a15",
      "value": "o",
      "empty_value": ""
    },
    {
      "name": "a16",
      "enabled": true,
      "label": "",
      "type": "string",
      "validate": {
        "required": true
      },
      "path": "a16",
      "value": "p",
      "empty_value": ""
    },
    {
      "name": "save",
      "enabled": true,
      "label": "Save",
      "type": "action",
      "path": "save",
      "validation_scope": {
        "name": "all"
      }
    }
  ],
  "fieldOrder": [
    [
      "firstName",
      "lastName"
    ],
    [
      "age",
      "birthday"
    ],
    [
      "likesBroccoli"
    ],
    [
      "address1"
    ],
    [
      "address2"
    ],
```

```
      [
        "favoriteColors",
        "gender",
        "secret"
      ],
      [
        "bio"
      ],
      [
        "cars"
      ],
      [
        "addcar"
      ],
      [
        "legoSets"
      ],
      [
        "addlegoset"
      ],
      [
        "_supler_static_-1345397749"
      ],
      [
        "id"
      ],
      [
        "a1",
        "a2",
        "a3",
        "a4",
        "a5",
        "a6",
        "a7",
        "a8",
        "a9",
        "a10",
        "a11",
        "a12",
        "a13",
        "a14",
        "a15",
        "a16"
      ],
      [
        "save"
      ]
    ]
  },
  "errors": [
  ]
}
```


A serialized form sent from the frontend to the backend when an action is invoked or the form refreshed is even simpler, as it only contains the values, without any meta-data on how the form should look like:


```
{
    "firstName":"Adam",
```

```
    "lastName":"",
    "age":10,
    "birthday":"2015-02-02",
    "likesBroccoli":false,
    "address1":"",
    "address2":"",
    "favoriteColors":[
        0,
        2
    ],
    "secret":"",
    "bio":"",
    "cars":[
        {
            "make":0,
            "model":1,
            "year":1990
        },
        {
            "make":1,
            "model":5,
            "year":2004
        }
    ],
    "legoSets":[
        {
            "name":"Motorcycle",
            "theme":1,
            "number":1924,
            "age":31
        },
        {
            "name":"Arctic Supply Plane",
            "theme":0,
            "number":60064,
            "age":1
        },
        {
            "name":"Princess and Horse",
            "theme":2,
            "number":4825,
            "age":7
        }
    ]
}
```


Moreover, Supler's frontend & backend are independent. They only communicate by the "json protocol" defined above. You could easily implement e.g. an alternative frontend.