DARWINO

UI Libraries

Table of Contents

Darwino UI Libraries Guide

Introduction

If Darwino is agnostic in term of UI, it provides some components to facilitate the creation of form based applications. These components can be reused within existing applications or used to create brand new ones. Moreover, because they are built on top of existing, standard technologies, they can be extended and/or augmented with other third party libraries.

Technology choices

As UI technologies are constantly evolving, Darwino will provide, in the longer term, support for several UI technologies. For web Technologies, we make a distinction between the core JS library and the CSS framework being used. As of today, here is the list of supported technologies:

1. ReactJS + Bootstrap.

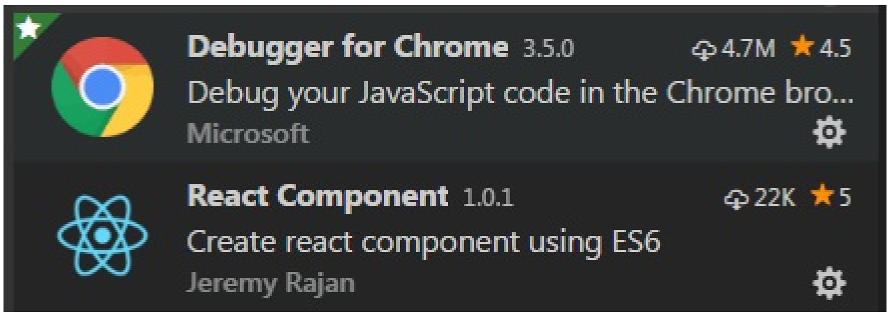
Installation - prerequisites

Darwino client side libraries require the latest node.js and npm to be installed. You can find more information on how to download and install this software here: https://nodejs.org/en/

Installing the tooling

If Eclipse can be used to develop modern web application, it is not currently part of the core platform but delivered through third party plug-ins. To provide the best developer experience, we use other dedicated tooling to edit the JavaScript code. In this tutorial, we use Microsoft Visual Studio Code. It can be downloaded from here: https://code.visualstudio.com/ As visual studio works well with the Chrome browser, make sure that the latest version of Google Chrome is properly installed. Finally, install the following plugins inside MS Visual Studio:

1. React Components
2. Debugger for Chrome



Debugger for Chrome 3.5.0 4.7M ★4.5
Debug your JavaScript code in the Chrome bro...
Microsoft

React Component 1.0.1 22K ★5
Create react component using ES6
Jeremy Rajan

Setting up the repository

The Darwino JavaScript assets are distributed through a private npm server serving all the packages scoped by @darwino.

To be able to consume them, you should instruct you local npm to use that server for @darwino. This is done by running the following command from the command line:

```
npm adduser --registry https://npm.darwino.com --scope @darwino
```

It will ask you for your darwino user/password and your email. Once done, you'll be able to consume all the darwino packages as described at: https://npm.darwino.com

Consuming the code

As of today, the JavaScript code is designed to be directly integrated as ES6 modules within the final application, and then bundled using a tool like Webpack. For performance and flexibility reasons, there is no ready to use single JavaScript file.

Common Code

The Darwino JavaScript UI libraries are available as ES6 modules through npm. They use a private repository identified named @darwino.

Available libraries

The Darwino npm server features a set of reusable JavaScript libraries:

- **@darwino/darwino** Core, pure Darwino JavaScript library that contain a set of useful utilities (access to the datastore, users, ...).
- **@darwino/darwino-react** Base react components used to create form based application. These components do not preclude any UI framework and thus should not be directly used. They provide the base behavior for derived components.
- **@darwino/darwino-react-bootstrap** Extension of the Darwino react components targeting Twitter boostrap. These are concrete components, ready to use.
- **@darwino/darwino-react-bootstrap-notes** Some specialized Darwino react-bootstrap components providing features and behavior similar to the IBM Notes rich client UI. These are mostly to be used in migration scenarios.

@darwino/darwino

@darwino/darwino-react

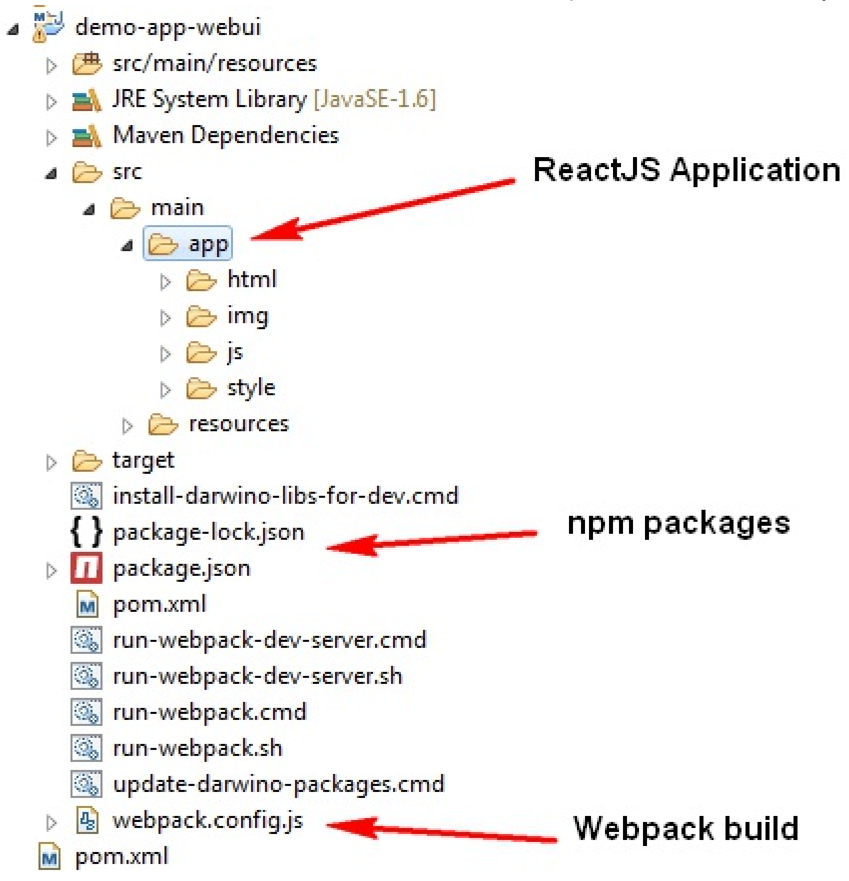
@darwino/darwino-react-boostrap

@darwino/darwino-react-boostrap-notes

Setting up the development environment

Project structure

When you create the project from the Darwino Studio and select react-bootstrap as the UI framework, you'll have the following folder structure:

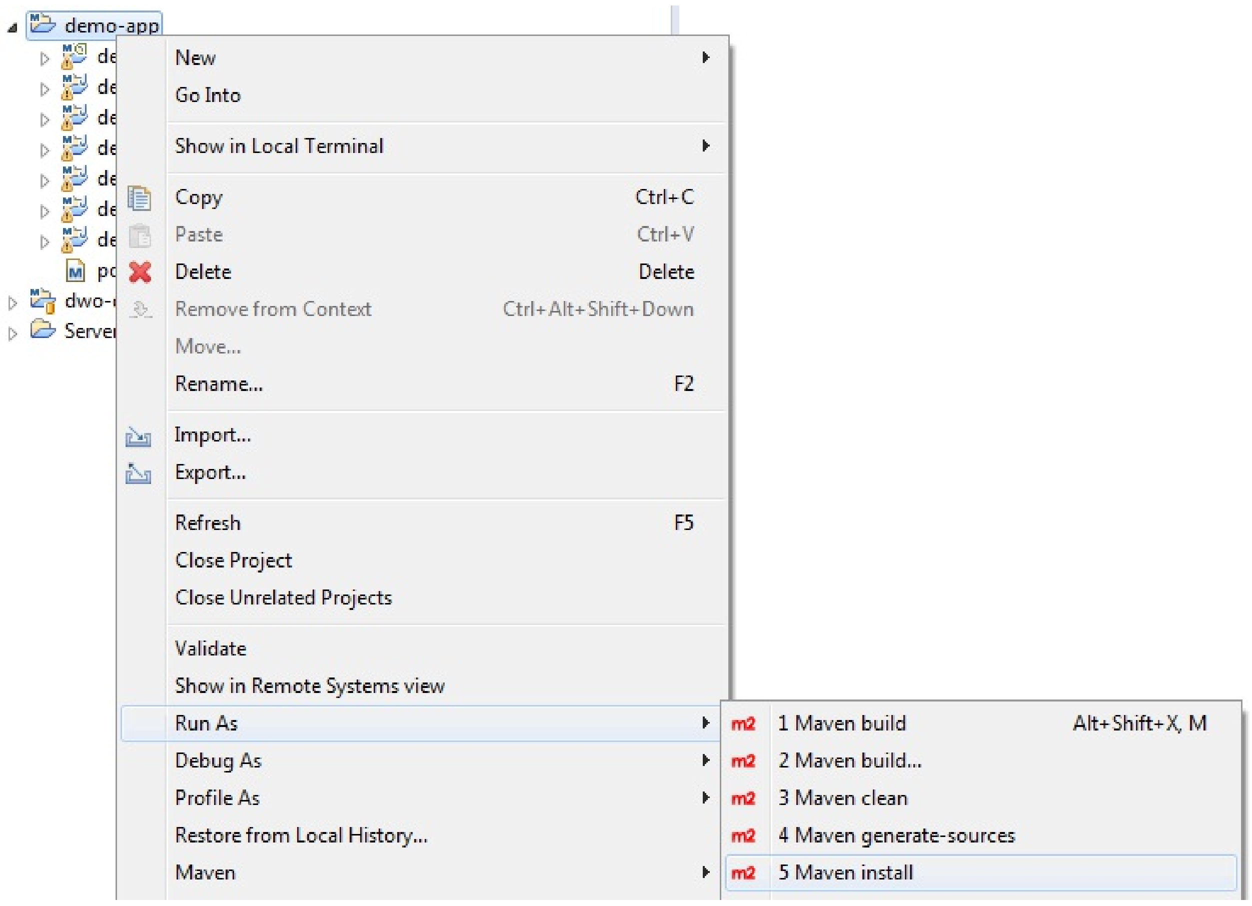


```
demo-app-webui
    src/main/resources
    JRE System Library [JavaSE-1.6]
    Maven Dependencies
    src
        main
            app
                html
                img
                js
                style
            resources
    target
    install-darwino-libs-for-dev.cmd
    package-lock.json
    package.json
    pom.xml
    run-webpack-dev-server.cmd
    run-webpack-dev-server.sh
    run-webpack.cmd
    run-webpack.sh
    update-darwino-packages.cmd
    webpack.config.js
pom.xml
```

All the files making the UI are located in the app directory. Thus, you can open this directory with MS Visual Studio Code, and start working on the UI.

Building and Running the code

The project is designed with continuous build in mind using maven as the main build tool. To build the whole project, including the Java and the JavaScript code, just execute a 'maven install' on the root project. Note that the very first build requires an internet connection and can take several minutes as it downloads a local copy of node.js, as well as all the project npm dependencies.

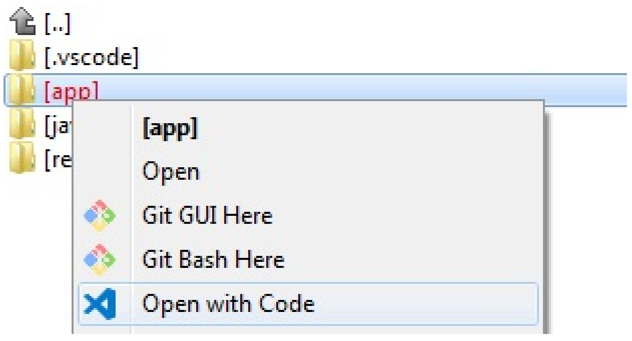


Assuming that Darwino is properly installed on the dev machine (darwino-beans.xml...), then the application can be added to a server and accessed using the following URL http://localhost:8080/demoapp This process generates a WAR file that can be deployed to any web application server, like any other Darwino application.

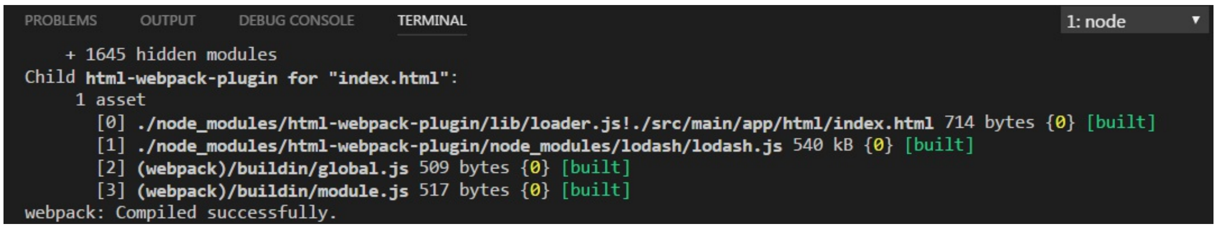
Running and debugging the code

Setting up MS Visual Studio

Darwino is taking advantage of Webpack and MS Visual Studio to run the client side application in debug mode. Here is how to enable that, assuming that MS Visual Studio is opened on the 'app' folder:



1. Open MS Visual Studio on the application 'app' folder
2. In MS Visual Studio, display the 'Integrated terminal view'
3. Enter the following command: `npm run dev`. This starts the Webpack development server. To not collide with the Java TOMCAT server, Darwino configured it to use port `8008`. Once launched, the terminal should display something like:



```
    + 1645 hidden modules
Child html-webpack-plugin for "index.html":
     1 asset
       [0] ./node_modules/html-webpack-plugin/lib/loader.js!./src/main/app/html/index.html 714 bytes {0} [built]
       [1] ./node_modules/html-webpack-plugin/node_modules/lodash/lodash.js 540 kB {0} [built]
       [2] (webpack)/buildin/global.js 509 bytes {0} [built]
       [3] (webpack)/buildin/module.js 517 bytes {0} [built]
webpack: Compiled successfully.
```

4. The application can now be accessed using the following URL: http://localhost:8008

The TOMCAT server should be running, as it still serves the data on port 8080. But this has a fgew challenges, as the 2 servers work are from different domains and do not share authentication:

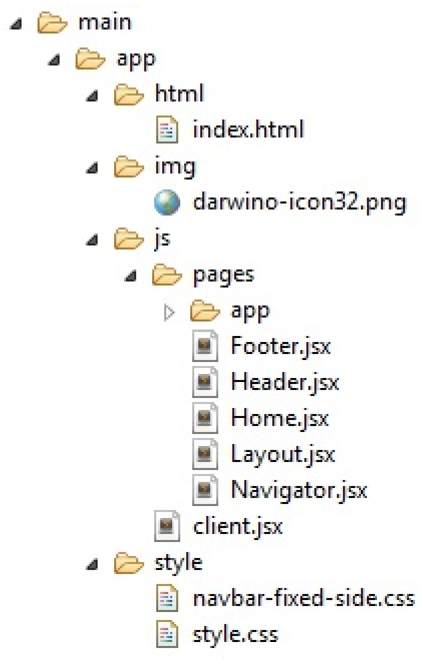
1. CORS should be enabled in web.xml Darwino provides a CORS filter that is enabled by default when the application is generated by the new project wizard. This filter can be removed, or customized, when going to production.
2. If the application requires authentication, you should open another window in Chrome and hit the TOMCAT server to get authenticated.

Debugging the application using Google Chrome

To debug the application, simply go to `Debug->Start Debug` in the MS Visual Studio menus. If there is no Chrome configuration, then just create one. Chrome should be launched and you should be able to debug your client code.

Building a React/Bootstrap application

The Darwino wizard generated a application skeleton with the following files:



```
main
  app
    html
      index.html
    img
      darwino-icon32.png
    js
      pages
        app
        Footer.jsx
        Header.jsx
        Home.jsx
        Layout.jsx
        Navigator.jsx
      client.jsx
    style
      navbar-fixed-side.css
      style.css
```

Let's look at the different files:

- index.html This is the main application page to launch. It contains the meta-data of the page, and a `<div>` that will host the whole ReactJS application
- client.jsx This is the main ReactJS page. It loads the JavaScript code, initializes all the libraries and instantiates the main layout component
- Layout.jsx This lays out the whole page using the bootstrap library. It creates a header, a footer and a body. It takes advantage of the responsive capability of the bootstrap library.
- Navigator.jsx
- Header.jsx This is a header, displaying by default the company brand and the current user
- Footer.jsx This is the footer, displaying by default the company brand and the current user
- Home.jsx This is the home page, as the first choice in the left Navigator

The contacts-react sample application

Darwino delivers a sample application to show case the different capability of the library. The source of the whole application is available on Github: https://github.com/darwino/darwino-demo/tree/develop/darwino-demo/contacts-react

This application is actually a port of a sample Notes/Domino application. The source NSF is provided in the root of the project. Darwino can be condifured to replicate the data between the NSF and the new Darwino application.

The following sub-chapters are describing how some common tasks can be achieved.

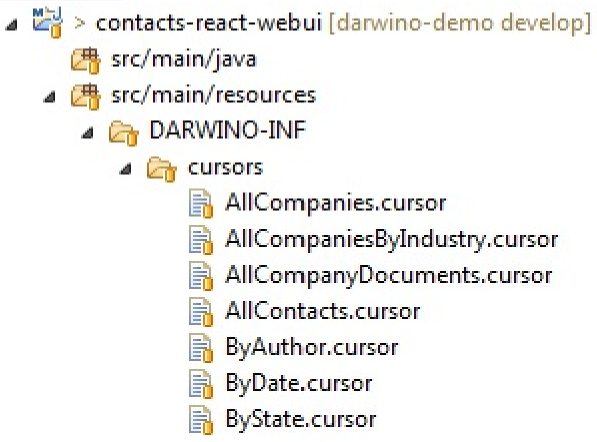
Adding a view

A view, in Darwino, is actually a query to the database. The developer will have to do a few tasks:

1. Define the database query
2. Create the UI for the view
3. Eventually, add the view to the navigator

Defining the query

The query can be defined dynamically in the UI control, or created as part of the application and then referenced by name. We're going to use the later, as it is a good idea to pre-define these queries. By default, such a query is a file of type `.cursor` located in the `DARWINO-INF/cursors` resources of your project. In the demo application, the queries are in the contacts-react-webui project:



```
contacts-react-webui [darwino-demo develop]
  src/main/java
  src/main/resources
    DARWINO-INF
      cursors
        AllCompanies.cursor
        AllCompaniesByIndustry.cursor
        AllCompanyDocuments.cursor
        AllContacts.cursor
        ByAuthor.cursor
        ByDate.cursor
        ByState.cursor
```

Each query is a JSON file containing the query containing the query definition. The name of the file is also the name used to reference the query when calling the REST service. Here is some sample content, from `AllCompaniesByIndustry.cursor`:

```
{
  "store": "companies",
  "extract": {
    "Industry": "industry",
    "State": {$USState: "$state"},
    "Name": "name",
    "form": "form"
  },
  "query": "{form: 'Company'}",
  "options": "DATA_MODDATES",
  "orderBy": "industry,state"
}
```

Creating the view UI

The UI can be done multiple ways, up to manually rendering. But Darwino provides a convenient UI Data Grid component that can be used to display a view, if this is the desired UI. This component provides many advanced capabilities, like dynamic sorting, categorization, collapsible children documents, totalization rows, responsive design... A typical view page is a JavaScript file containing 2 components: the grid component with eventually some customization, and the page component. Isolating the grid in it own component allows its reuse in other parts of the

UI (popup showing the view, embedded view...).

The AllCompanyByIndustry.jsx defines such a view. The most important parameters are the grid ones, defined there as `defaultProps`.

Adding the view to the navigator

There are 2 things to do:

1. Define the route path to reach the view This defines the URL used by the router to access the view. It associates a path with the page component to display.

   ```
   <Route exact path="/app/allcompaniesbyindustry" component={AppAllCompaniesByIndustry}></Route>
   ```

2. Add the view to the UI navigator This is done in the Navigator.jsx with a `NavLink` component.

   ```
   <NavLink to="/app/allcompaniesbyindustry">By Industry/State</NavLink>
   ```

Defining the query condition

Aquery points to a database and a store. If nothing else is provided, then all the documents from the store are retrieved

Adding a query condition

Aquery condition is defined by the `query` attribute in the query definition:

```
{
  "query": "{firstName: 'Jones'}",
}
```

This uses the standard Darwino query language and can be as complex as needed. Some examples are available in the playground: https://playground.darwino.com/playground.nsf/DarwinoDbSnippets.xsp#snippet=Queries_-_SQL_Optimization/

Selecting Notes/Domino documents based on a form

When documents are replicated from Notes/Domino, then the Form field contains the form and a query can be applied as bellow:

```
{
  "query": "{form: 'Company'}",
}
```

Retrieving children documents

Children documents can be retrieved using the `parentid` attribute. Generally, this parent ID should not be hard coded in the query itself, but passed dynamically from the view component. This is done in the UI Grid component itself. The `Company.jsx` form shows how to display an embedded view using the current document as the parent:

```
    <CursorGrid
      databaseId={Constants.DATABASE}
      params={{
        name: "AllCompanyDocuments",
        parentid: this.state.unid
      }}
      showResponses={true}
      columns={[
        {name: "Title", key: "Title", sortable: true, sortField: 'title'},
      ]}
    />
```

{% raw %}

Defining the view columns

ADarwino grid is displaying the data in a tabular form, using columns. The column can be

Extracting the column data

If nothing is specified in the query itself, then the whole content of the documents is extracted and returned as part of the data. This might be ok, but a developer might want to define what data should specifically be retrieved:

1. To minimize the amount of data transferred from the server to the client
2. To calculate some result on the server side, and make them simple to consume by the client

When columns are being defined, then the returned data is a JSON object with one entry per column. Columns are defined in the query object through the `extract` attribute. The attribute key is the column name, while the value is the expression to be evaluated for that columns. The expressions are evaluated for every single document:

```
{
  "extract": {
    "Industry": "industry",
    "State": {$USState: "$state"},
    "Name": "name",
    "form": "form"
  },
}
```

Asimple name indicates the name of a field at the root of the document.

Using Notes/Domino @formula language

Darwino is equiped with a @formula interpreter that can be used to calculate data based on the JSON document. $atFormala is the function used to evaluate an actual formula:

```
"CommonName": {
  "$atFormula": '@If(LastName="";"Misc";"  "+@Left(@ProperCase(LastName);1))'
},
```

Adding a form

Aform is simply a ReactJS component used to edit the data coming from a REST service. The data is handled through Redux, and more specifically through Redux Form. To ease the process of creating form, Darwino provides some easy to use components:

1. Abase class for the form, handling the communication with the server, calculated fields and other helpser
2. Some edit components doing the data binding with the document

Creating a Form

Aform component should inherit from

```
import {DocumentForm} from '@darwino/darwino-react-bootstrap'
class Company extends DocumentForm {
  ...
```

Form Fields

Fields can easily be added to a form, following the React Redux instructions. Darwino provides some easy to use helpers that brings more features with the editable components as well as an optional label.

Adding a component

The main component, `Field`, is provided by React Redux. Underneath, it uses some Darwino helpers providing the extra features, the proper CSS formatting (ex: bootstrap), the mode management (readonly, ...) and a label. Here is an example of a simple edit field:

```
<Field name="name" type="text" label="Name" component={renderText} disabled={disabled} readOnly={readOnly}/>
```

- name: is the attribute name in the JSON
- type: is the HTML input type
- component: the Darwino helper component to use
- disabled: indicates if the component is disabled, meaning that it cannot be used to edit the data
- readOnly: indicates if the component is readonly, meaning that it only displays some static text

Darwino provides a whole set of components beyond the simple edit box, like checkboxes, radiobuttons, ..

See: `extras/AllFields.jsx` for more information

Radio buttons, comboboxes and other lists

Some components require a list of values, whenever it is static or dynamically computed. In this case, the list is passed through the options attribute. It can be a simple array of values (the value and the label are then identical), or it can be an array of objects with a value and a label. Note that the combobox also features an `emptyOption` to display a label for an empty value.

Static examples:

The options attribute is just the list. Note that it is better if the list is statically defined in the JS file, so it is *not* recreated every time the component is rendered:

```
options={["M","F"]}
```

or

```
options={[
  { value: "M", label: "Male"},
  { value: "F", label: "Female"}
]}
```

See: `extra/AllFieldsRadioButtons.jsx` or `extra/AllFieldsSelect.jsx`.

Dynamic examples

In that case, the list is read from the database. Note that the list should be read just once, and asynchronously to provide the best user experience. The values could be stored in the main component state (the easiest) or in a Redux Store (requires more work, but can avoid a full page refresh when the values change).

The example bellow shows a list being read once when the component is created, and then another list reloaded when a value changes (using `calculateOnChange`)

See: `extra/DynamicSelect.jsx` .

Dealing with multiple value fields

Multiple value fields in Darwino are stored as JavaScript arrays in the JSON document, but are represented in edit boxes as a simple list of values separated using separator character (a coma by default). Checkboxes also support multiple values. An edit component is made multiple by adding the following attribute: `multiple={true}`

See: `extra/AllFieldsMultipleValues.jsx` .

Pickers

Darwino is equiped with a pluggable picker API, along with default picker implementation.

Adding a picker to a field

Apicker is simply a `renderValuePicker` component assigned to a field:

```
<Field name="myvalue" type="text"
  component={renderValuePicker}
  label="Select a value"
  disabled={disabled} readOnly={readOnly}
  picker={this.picker}
  pickerTitle="Select a value"/>
```

This creates a textbox with a button to let a user choose a value, or a list of values when the filed is multiple. There are 2 important parameters:

- picker The component that let a user choose a value. This is a function that should render a piece of HTML to be presented in the dialog. It is generally using one of the predefined pickers.
- pickerTitle The title of the modal being displayed

The Darwino API is very flexible and allows custom implemetations to be defined. But the core library provides the following ones:

Simple picker components

List of static values

```
picker() {
    return (
        <ListPicker
            values={["one", "two", "three", "four"]}
        />
    )
}
```

List of static values with a label

```
picker() {
    return (
        <ListPicker
            value="value"
            label="label"
            values={[
                {value:1, label:"one"},
                {value:2, label:"two"},
                {value:3, label:"three"},
                {value:4, label:"four"}
            ]}
        />
    )
}
```

User picker

There is a predefined user picker that takes advantage of the user service to select users.

```
picker() {
    return (
        <UserPicker
        />
    )
}
```

Database pickers

Use a list coming from a database query

The query returns a JSON document withe 2 fields: Name & State. Name is used as the value to store, while the label is computed from the name and the state:

```
picker() {
    let jsc = new JstoreCursor()
        .database(Constants.DATABASE)
        .store("_default")
        .queryParams({name:"AllCompanies"})
    return (
        <ListPicker
            value="Name"
            label={(e) => e.Name + " (" + e.State + ")"}
            dataLoader={jsc.getDataLoader()}
        />
    )
}
```

Use a simple grid with data from a database query

```
picker() {
    let jsc = new JstoreCursor()
        .database(Constants.DATABASE)
        .store("_default")
        .queryParams({name:"AllCompanies"})
    return (
        <GridPicker
            value="Name"
            dataLoader={jsc.getDataLoader()}
            columns={[
                {name: "Name", key: "Name", sortable: true, sortField: 'name'},
                {name: "Industry", key: "Industry", sortable: true, sortField: 'industry'},
                {name: "State", key: "State", sortable: true, sortField: 'state'}
            ]}
        />
    )
}
```

For more example, please look at `extras/Pickers.jsx`.

Computed Fields

Computed fields are complex in ReactJS but Darwino makes them simple.

Type of fields

Darwino maintains 2 set of fields: the real fields from the JSON document, and the calculated ones, stored apart. The form base component provides the `getFieldValue(name,def)` function to access either set: it first looks for a value in the field set. If it is not available, then it looks in the computed fields set.

Note: only real field fields can be bound to an editable component. Computed fields must be displayed using the `ComputedField`. See: `extra/AllFieldsComputed`.

Calculating/initializing fields

The base `Form` class features a set of methods used to initialize default field values or calculate compute fields

Field initialization

- defaultValues(json) Called once when a new document is created. The json parameter contains the current fields for the document. The function can modify this object to initialize any field. Once processed, the document in the redux store will have these values

Computed fields

Computed fields can be calculated once, whenever the document is loaded/created, or every time it is needed:

- calculateOnLoad(values) This is called once when the document is loaded/created. The calculated values must be returned in the values parameter.
- calculateOnChange(values) This is called every time there is a change in the document data (ex: a user typed a character in a field). This means that the execution of this function must be as fast as possible to not degrade the user experience.

Computed fields & Subforms

Subforms can also contribute default values and computed fields, similarly to the forms. The same methods can be implemented.

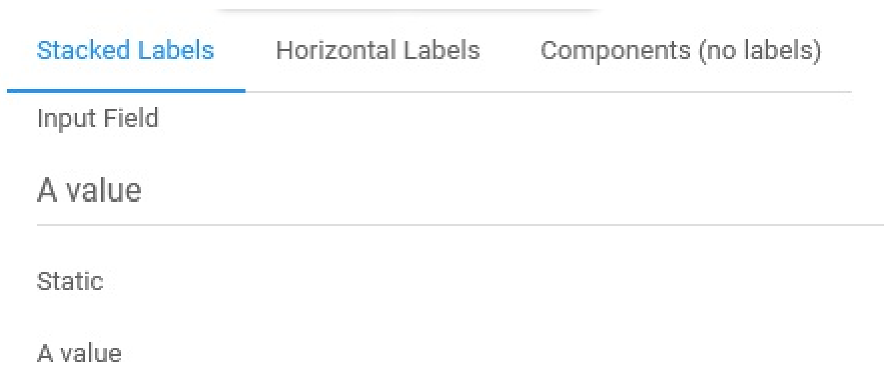
See: `extra/CCAddress.jsx`

Computed fields optimization

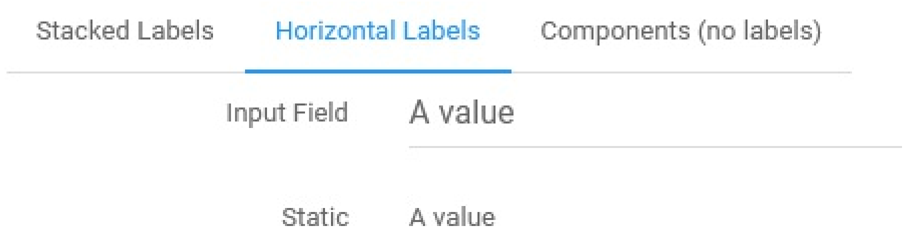
The core Darwino focuses on making the use of computed fields very easy, at the expense of doing the calculation on each change, and then re-rendering the whole when a calculated value changed. This should not be a problem for most applications, although different strategies can be used to optimize both the calculation and the rendering, typically acting at the redux store level. This is beyond the scope of this library for now.

Field Label Layout

The renderXXXX components have a label that can be on top of the component or on the left side. This is driven by the attribute `horizontal`.

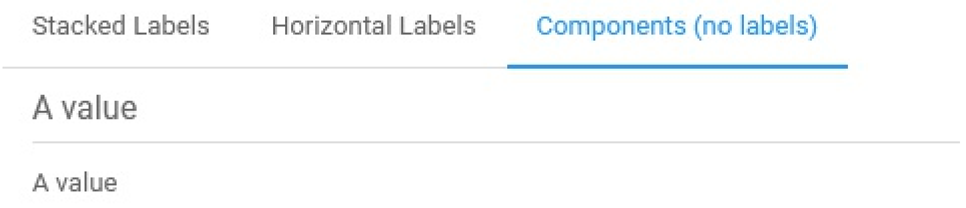


Stacked See: `extras/FormLayoutStacked.jsx`



Horizontal See: `extras/FormLayoutHorizontal.jsx`

Plain Components (no label) To completely avoid displaying labels, use the FieldXXXX component instead. They behave the same than the renderXXXX without being decorated with a label.



See: `extras/FormLayoutComponents.jsx`

Read Only Forms and Fields

Fields can have 3 states:

- Editable This is the normal state, allowing a user to edit the value
- Disabled The editable control is displayed with the value, but it is not editable
- Readonly The editable control is replaced a piece of static HTML, generally a text

Controlling the display mode

The component mode is controlled via the `disabled` & `readOnly` properties. They have to be passed explicitly to every single control. When absent, hen the control is in normal editable mode.

Form display mode

Aform can carry flag for disabled or readOnly. These flags are part of the form state, but are accessed using getters:

- isReadOnly()
- isDisabled()
- isEditable()
- getMode()

By default, the form is in a DEFAULT_MODE, which means that the actual mode (editable or disabled) is deduced from the document security. If the document is read only, then DEFAULT_MODE means disabled. But this value can be forced using the `setMode()` method.

Typically, component mode is set to every component in a form like this:

```
render() {
    const readOnly = this.isReadOnly();
    const disabled = this.isDisabled();
    ...
    return (
        ...
        <Field name="firstname" type="text" component={renderText} label="First Name" disabled={disabled} readOnly={readOnl
```

See: `extras/AllFields.jsx`.

Configuring the navigator

The Darwino library uses the React Router library to map the URLs to pages and handle the navigation (back button, ...), while being in a single page application.

Navigator entries

In the demo app, the left side navigator is configured in `Navigator.jsx`. Each entry is provided through a `NavLink` component, which set the routing path and also highlight the current entry:

```
<NavLink to="/app/allcompanies">All Companies</NavLink>
```

Router

The React Router has a `Switch` component to select what component to render based on the current URL. This is done in the `Layout.jsx` component. Every route (e.g. component to display base on a URL) is defined in a Route component, like bellow:

```
<Switch>
  <Route exact path="/app/allcompanies" component={AppAllCompanies}></Route>
  <Route exact path="/app/company/" component={AppCompany}></Route>
  <Route exact path="/app/company/:unid" component={AppCompany}></Route>
```

The example above shows 2 different kind of routes:

1. allcompanies, routing to a view, with no parameter. These URL
2. company and company/:unid, routing to a form with an optional single parameter `unid`. When the unid is available, then the corresponding document will be loaded. Else, a new document will be created.

Interacting with the server

The Darwino UI library provides a few built-in capabilities to interact with the server side.

Darwino service URLs

Darwino has a very convenient capability that make the use of service URLs a lot easier. By default, when a service path starts with `.darwino-`, it can also be called using `$darwino-`. In the later case, the URL is transformed back to `/.darwino-`, regardless of the relative position of `$darwino-`. This means that `$darwino-` can be used without worrying about the current browser URL. On other words, `/$darwino-`, `/x/y/$darwino-`, `/a/b/c/$darwino-` are all equivalent when calling the server, and reach the exact same service.

Resources and data servers

In a normal production mode, the UI elements (JS files...) and the data will be served by the same JEE application. But, during development, while the data will still be served by the J2EE server, we'd like to use a Webpack development server. Even if the 2 servers run on the same machine, they will be using a different port and thus will not share the authentication, while AJAX calls will faces cross domain issues.

When developing using TOMCAT, the JEE serverport is 8080 while the Webpack server port is 8008 (defined in `webpack.config.js`)

Authentication

The Webpack dev server does not authenticate users, while the real JEE server might. To make it work, the user will have to manually authenticate in another browser tab. Under the hood, the Darwino library ensures that his authentication is used by setting the request credentials to "include" (see dev.js) when it detects the development server.

Cross domain requests

To support this mode, the CORS filter have to be enabled in the JEE `web.xml` (this is the default when using the Darwino Studio wizard):

```
<!-- CORS should be applied before the authentication else the OPTIONS preflight can be blocked -->
<filter>
    <filter-name>Cors</filter-name>
    <filter-class>com.darwino.j2ee.servlet.cors.CORSFilter</filter-class>
    <init-param>
          <param-name>cors.allowed.methods</param-name>
          <param-value>GET,POST,PUT,DELETE,HEAD,OPTIONS</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>Cors</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

Moreover, as the JEE path is different, that path must be initialized using the DEV_OPTIONS class. The Darwino runtime will then query that class for the URL. The initialization happens very early, in `client.jsx`:

```
// App rendering
import {DEV_OPTIONS,initDevOptions} from '@darwino/darwino';

// Redux dev tools
if(process.env.NODE_ENV!="production") {
    initDevOptions("http://localhost:8080/contacts-react/")
    ...
}
```

REST Services

Of course, any existing JavaScript capability can be used to call REST services. In particular, the new `Fetch` capability. But Darwino makes it both easy to create a service on the server side, and to call it from the client.

Calling a REST service

Darwino provides some utility in `Fetch.js`. This utility is a thin wrapper on top of the browser `Fetch` function that:

- Seamlessly handles the development server URL and credentials differences
- Treat request with status code not in 200-299 as error and throw an exception
- Have some simple utility functions, `fetchJson` & `fetchText`, that parse the response and return the expected data type in a promise

See: `extras/ServicesRest.jsx`

Developing a REST service

Such a REST service has to be done in Java on the server side. There are a few classes to define:

The REST service class

This is the class that implements the actual service. The service() function gets a context and should return a value, or an error, on execution. One instance of this class will be created per service execution. The instance will be discarded after the service is executed.

see: `contacts-react-shared/../AppInformationRest.java`

The REST service factory class

This class creates the REST service instance for a particular request, by matching the URL parts. Atypical factory will match multiple services with multiple URLs. Afactory has to be registered at the application level, see: `AppBasePlugin.java`

see: `contacts-react-shared/../AppRestServiceFactory.java`

Micro services

Micro services are a simplified form of stateless services. Amicro service consumes some JSON content as parameters and also produces JSON as a result. This is that simple. Micro services can be written in different languages, although the contacts-react sample shows how to write them in Java.

For more information: https://playground.darwino.com/playground.nsf/ServicesSnippets.xsp

Creating a micro service

Amicro service implements a Java interface called `JsonMicroService`. This interface has one method called when the service is invoked. It has a context parameter that contains all the request information.

Here is a basic `HelloWorld` service, that takes a greetings parameter and returns a formatted message:

```
public void execute(JsonMicroServiceContext context) throws JsonException {
    Session session = DarwinoContext.get().getSession();
    JsonObject req = (JsonObject)context.getRequest();
    String greetings = req.getString("greetings");
    JsonObject result = JsonObject.of("message",StringUtil.format("Hello, {0}. {1}. It is {2} here, on the server!",
                session.getUser().getCn(),
                greetings,
                DateFormatter.getFormat("DEFAULT_TIME").format(new Date())));
    context.setResponse(result);
}
```

- `Session session = DarwinoContext.get().getSession()` Get the session database for the current user. With this session, the developer can access the data on the behalf of the user.
- `String greetings = req.getString("greetings")` Read a parameter from the the requester. In that case, the request is made of a JSON object with on attribute called 'greetings'
- `context.setResponse(result)` Send the JSON result to the caller

See: `HelloWorld.java`

Registering a micro service

Micro services have to be registered using a factory. Aunique name has to be assigned to every single micro service. This name will be used by the client when invoking the service.

See: `AppMicroServicesFactory.java`

Consuming a micro service

Darwino provides a very convenient helper `MicroServices`, to invoke the services. Simply pass the name, the parameters and fetch the result. Then, either get the result or handle the error

```
new MicroServices()
  .name(valid?"HelloWorld":"fake")
  .params({greetings: "I am the React client calling you"})
  .fetch()
  .then((r) => {
    this.setState({
      error: false,
      result: JSON.stringify(r,null,2)
     })
    })
   .catch((e) => {
     this.setState({
       error: true,
       result: e.message+"\n"+e.content
     })
    })
```

See: `extras/ServicesMicro.java`

Advanced capabilities

Custom query functions

The Darwino query/extration language allows the creation of custom functions. !! If such functions can be used in both query and data extraction expression, bear in mind that they might slow down the query as it cannot be fully processed by the underlying SQL database.

As an example, the demo application defines the '$USState' function that returns the label of state for a given code.

Here is how it is implemented

Creating the function

As the function could be executed on both server and mobile environment, it is defined in the shared project. The name of the function is `OpUSState`, which inherits indirectly from `com.darwino.commons.json.query.nodes.Node`. The main method is `execute`, which uses a context to calculate the resulting value. Many examples of such functions or operators are provided as part of the Darwino source code.

Registering the new function

The new function has to be registered using a factory provided through a standard Darwino extension. In the example, the factory is called `AppQueryExtension`. It currently only defines one function but it can define as many desired. The factory itself should be registered by the application plugin (`AppBasePlugin` in the example):

```
if(serviceClass==QueryExtension.class) {
    extensions.add(new AppQueryExtension());
}
```