# Signum Framework Tutorials Part 5 – Southwind React

## About Signum Framework

Signum Framework is an application framework for making data-centric windows and web applications. It promotes a code-first workflow and is focused in composability, to share code between projects.

## About this series

In this series of tutorials we will work on a stable application: Southwind.

Southwind is the Signum version of Northwind, the well-known example database provided with Microsoft SQL Server.

In this series of tutorials we will create the whole application, including the entities, business logic, windows (WPF) and web (React) user interface, data loading and any other aspect worth to explain.

If you want to know more about the principles of Signum framework look at the previous tutorial:

* [Signum Framework Principles](http://www.codeproject.com/KB/linq/SignumFramework.aspx)
* Signum Framework Tutorials Part 1 – Southwind Entities
* Signum Framework Tutorials Part 2 – Southwind Logic
* Signum Framework Tutorials Part 3 – Southwind Load
* Signum Framework Tutorials Part 4 – Southwind Windows

In this tutorial we will close the circle and we will show how to create a Web user interface using TypeScript, React, Bootstrap and Signum.React.

We will try to keep a balance between theory and practice, making short breaks to explain important concepts as we need them.

### Prerequisites

Signum.React helps you build a SPA (Single Page Application) using C#, TypeScript, React and Bootstrap.

* **C#** is a general purpose, statically typed, object-oriented programming language that has been evolving into a more functional style after C# 3.0. (Need to know: Generics, LINQ, …)
  + **.Net:** A general purpose framework for building applications that run on Windows and has been evolving into becoming multi-platform with .Net Core. (Need to know: Web.API, …)
* **TypeScript** is a free and open-source programming language developed and maintained by Microsoft. It is a strict syntactical superset of ES6/JavaScript 2015, and adds optional static typing to the language. (Need to know: Type declarations, Generics, non-nullable types…)
  + **JavaScript** is a programming language used to make webpages interactive. It is what gives a page life—the interactive elements and animation that engage a user. (Need to know: how ‘this’ works, prototypal inheritance)
  + **ES6/JavaScript 2015** is the new version of JavaScript that introduces many new features. (Need to know: classes, lambdas and this, spread operator, ES6 modules, promises, async/await)
* **React** is a declarative, efficient, and flexible JavaScript library for building user interfaces build by Facebook. Your components tell React what you want to render – then React will efficiently update and render just the right components when your data changes. (Need to know: props/state, Virtual DOM, component lifecycle)
  + **JSX:** Extension of JavaScript to allow defining snippets of HTML-like expression inside of JavaScript. Usually use together with React and compiled-down to JavaScript. JSX files have the extension .jsx or .tsx, when used together with TypeScript. (Need to know: JSX syntax, spread operator)
* **Bootstrap** is a free and open-source front-end web framework for designing websites and web applications. It contains HTML- and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions. (Need to know: grid, forms)
  + **react-bootstrap:** While bootstrap is mainly a css framework, it also contains some JavaScript that does direct DOM manipulation using jQuery. This is contrary to the declarative philosophy of react and creates incompatibilities. In the current moment, we are using react-bootstrap to bridge these two technologies.

A basic knowledge of C#, JavaScript/ES6, TypeScript, React and Bootstrap are necessary to be an effective developer of Signum.React and a prerequisite for this tutorial.

### Introduction

Signum Framework is a cohesive framework where all the pieces are design to work nicely together. We tried to keep a consistent API and a philosophy between Signum.React and Signum.Windows, simplifying moving from one to the other.

The two biggest pieces of Signum.React, just as in Signum.Windows, are the edition controls (created using property components) and the search windows. There is also a similar Navigation module, Finder module, EntitySettings, OperationSettings, etc…

On the other side, Southwind.React, as opposed to Southwind.Windows, runs in the browser so we will need to split out code in two worlds: Server-side (C#, LINQ) and Client-side (TypeScript, React and Bootstrap). Signum.React tries to keep these two worlds connected by converting entities to JSON, generating TypeScript definitions and providing metadata information.

Being this said, we also try not to force the hosting platform (TypeScript / React) with unnatural conventions. We want to be good citizens in both Windows and React, allowing the programmer to express their creativity whenever the resources of our libraries don’t fit the requirements.

### Connecting Client and Server

Signum.React tries to keep your client code (written in TypeScript and React and running in the client browser) connected with your server code (written in C# and running in your private infrastructure).

There are some important benefits in having these two worlds connected:

* Thanks to the metadata transmitted between client and server, the UI Controls have excellent default behavior (Buttons enabled, combo-boxes are filled, auto-completes configured…) this drastically reduces the amount of code that needs to be written and maintain.
* Encourages developers to become full-stack, empowering them to complete tasks by themselves, reducing interdependencies in the development process.
* Facilitates having a consistent behavior between the database and the user interface, since it was typically build by the same person.
* By having statically typed interfaces for each entity, changes in the database that could affect the UI are detected at compile-time.
* Because there is much less code to maintain, and is strongly typed, it’s easier to keep it up to date with the continuous changes in the underlying NPM packages, even making it possible to do radical technological changes like Web -> React.
* Empowers admin users to have an understanding of the database schema to make reporting, templates, etc... Also facilitating communication with developers.
* Since the dependency is from Signum.React to Signum.Engine and not the other way around, the server code remains agnostic from the UI and is still completely possible to have alternative UIs like Signum.Windows or any future technology.

Let’s see how Signum.React makes this connection possible.

* **Allowing Entities, Lites, MLists to be serialized in JSON** (EntityJsonConverter, LiteJsonConverter, MListJsonConverter).
* **Generating TypeScript code at Compile-time** from the entities to keep the client side strongly typed.
* Providing **Metadata information** by the server including entity names, property names, translations, implementations, security, etc… giving the React UI the necessary information to automatically configure the default behavior of the property components, or even completely generate an automatic UI (DynamicComponent).

**TypeScript is a compile-time illusion:** Coming from C#, it could be surprising for you that a metadata service is necessary. Couldn’t this information be inferred from the generated TypeScript code? Unfortunately no. All the type information that TypeScript uses during compilation is gone at run-time, where only dynamically-typed JavaScript code is left. Moreover, most of the information provided by this metadata service is not available by looking at the types themselves: Translations, authorization rules, etc…

We use ASP.Net Web.API and Json.Net to send the Entities, Models, or just plain DTOs from the server to the client and back.

Once in the client, the entity objects, now converted in JSON objects, preserve some run-time information from the original C# one:

* The ‘**Type’** property (string) contains the clean name of the entity type.
* The ‘**modified’** property (boolean) contains whether the entity has changes that should be saved back in the database (is dirty).
* The ‘**isNew’** property (boolean) contains whether the entity was recently created and needs to be inserted (instead of updated) when saved to the database.
* The ‘**id’** property contains the primary key of the entity, typically a number (for int, long, etc…) or a string (for string, GUID…)

For each C# Entity class a new TypeScript Entity interface is going to be auto-generated, allowing strongly typed access to all the properties of the entity.

The methods, however, are lost in translation because it is non-trivial to convert arbitrary C# code to JavaScript/TypeScript. Even worst, we cannot define these methods manually on the client side, because we do not set/remove the JavaScript ‘prototype’ when receiving/sending the entities to the client side.

In practice this is not a big deal, just some methods that in the server-side are instance methods (customer.ToLite()) here are external functions (toLite(customer)).

## Making Contact

The first thing to do is to load Southwind.React, the last unloaded project.

In there, we can see many files. Some of them belong to the server side (.cs, .cshtml, .config, .asax) while others will be used to bundle the client-side (.js, .ts, .tsx, .css, .json). Typically, these files are organized by application module, not by technology, so it can be a little confusing at the beginning but then it helps to keep related code close to each other. For now, let’s divide the files for simplicity:

#### Server-Side global files

* **packages.config**: Contains the Nuget references of the project.
* **Web.config**: This XML file contains the configuration for ASP.Net MVC and Web.API. Here we can find the connection strings, and some basic configuration. *Note: More advanced configuration is typically found in ApplicationConfigurationEntity and can be change at run-time*.
* **Global.asax/.cs:** Contains the start-up code for the server side. Including:
  + Server-side URL routing. Typically, everything is redirected to the single .cshtml except routes starting with /api/.
  + Starting server application (Starter.Start)
  + Initializing server application to fill some mandatory caches
  + Starting background tasks (Processes, Scheduler Tasks, Async email sender)
  + Starting the servers’ classes, to enable Controllers defined in Extensions
  + Configuring modules in the server-side, like Omnibox
  + Determining the default culture for a request
* **ServerSouthwind.svc/ServerSouthwindTransfer.svc:** Optional, WCF services for Signum.Windows.
* **Controllers/HomeController.cs:** The only ASP.Net MVC controller, returning the single .cshtml view for the SPA.
* **Views/Home/Index.cshtml**: The only view, a very simple HTML page that loads the latest version of the bundled JavaScript code and adds some necessary polyfills.

#### Client-Side global files

* **package.json**: Contains the references to the NPM packages, like React, moment and a never-ending list of always-changing dependencies.
* **tsconfig.json:** Contains the configuration for the TypeScript compiler.
* **webpack.config.js:** Contains the configuration for the main bundle, containing all the Signum.React/Signum.React.Extensions code together with your own custom code. The entry point is Main.tsx.
* **webpack.config.dll.js:** Contains the configuration for the vendor bundle, containing the code of react/moment and all the other third-party NPM packages. The entry point is vendor.js.
* **App/:** This folder contains your custom TypeScript/React code for the client-side and the custom Web.API controllers needed for your SPA.
  + **vendor.js:** Contains references to the third-party NPM packages that should be bundled together (mandatory dependencies).
  + **Main.tsx:** Entry point of the SPA, has many responsibilities:
    - Configure moment.js, numbro.js and React Widgets.
    - Configure auto-login with cookies and sharing session between tabs.
    - Configure error handling
    - Start all the used Client modules defined in Framework/Extensions and your current application.
    - Configure custom client-side URL routings using react-router (many routes are defined by the Client modules).
    - Render the SPA in ‘reactDiv’ element.
  + **Layout.tsx:** Containsthe main component that will wrap any page (master page), including the Navbar, SideBar and Footer.
  + **Home.tsx:** Welcome page, typically loading the default Dashboard for a user or showing a bootstrap Jumbotron.
  + **NotFound.tsx:** Fallback page when no client-side route matches the URL.
  + **site.css:** Contains the styles for the whole application. It is recommended to keep component-specific CSS rules in isolated .css files.

#### Module-Specific files (Southwind)

Most of the work of creating an application using Southwind.React is going to be in the sub-folders of App. Each sub-folder represents a cohesive module that is usually represented by one namespace of entities.

In our example, we have only one module (Southwind) but in real applications is usual to have many. Let’s see what files we have there:

* **SouthwindClient.tsx:** Entry point of the client modules. Typically, associates entity types with custom react components, configures operations or creates new routes to custom page components.
* **Southwind.Entities.t4s:** Tells Signum.TSGenerator that we want to generate here the TypeScript definitions for the namespace “Southwind.Entities”. This custom extension is a mixture of .t4 (Microsoft template language) and .ts (TypeScript file).
* **Southwind.Entities.ts:** This file is generated by Signum.TSGenerator, and contains the type declarations for your entities and other assets defined in the Entities assembly.
* **CatallogController:** Server-side Web.API controller with custom actions necessary for this module. Signum already provides a battery of general-purpose controllers for making queries, retrieving entities or executing operations, so often no controller is necessary.
* **Templates:** Folder containing all the custom react components for your entities. They are usually take a TypeContext<YourEntity> as props and use property components (EntityLine, ValueLine, …) internally, but you can also write any custom react code that you need to create an awesome UI experience. If no template is defined, DynamicComponent/DynamicViewComponent will create one automatically from the meta-data.

#### Signum.TSGenerator

Signum.TSGenerator is a MSBuild task that generates, at compile-time, typescript definitions for your entities. Let's see how it works:

* The generation process is controlled by Signum.TSGenerator, registered as a task in Southwind.React.csproj.
* It uses conventions to find the Southwind.Entities assembly, and explores all the types using reflection in the “Southwind.Entities” namespace.
* When entities in “Southwind.Entities” have references to other namespaces or assemblies, is able to find, again using conventions, the appropriate auto-generated TypeScript files and add the necessary ‘import’ statements.
* Any TypeScript code that you write in the .t4s file, will also be included at the beginning of the generated file, at the very top. You can use TypeScript declaration merging to extend interfaces.

#### Type<T>

If you take a look at the generated code, like Southwind.Entities.ts, you will see often the following pattern:

export const ProductEntity = new Type<ProductEntity>("Product");

export interface ProductEntity extends Entities.Entity {

    Type: "Product";

    productName?: string | null;

    supplier?: Entities.Lite<SupplierEntity> | null;

    category?: Entities.Lite<CategoryEntity> | null;

    quantityPerUnit?: string | null;

    unitPrice?: number;

    unitsInStock?: number;

    reorderLevel?: number;

    discontinued?: boolean;

}

Let’s analyze this code:

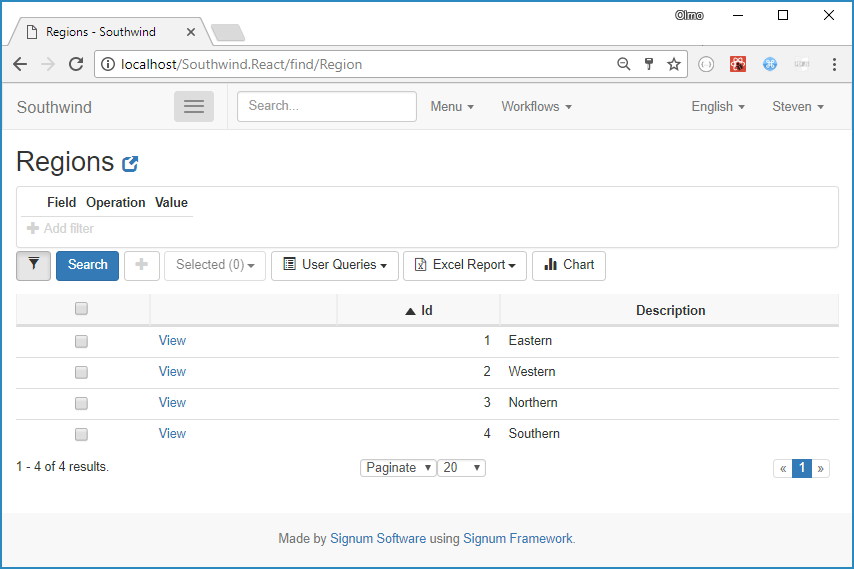
* The Entity is defined as an interface, not a class. In TypeScript interfaces can also be used to be implemented by classes, but most of the time they just define the shape of a JavaScript/JSON object. As we said, Entities will be sent to the client side as pure data objects with no behavior or prototype.
* Two elements are exported with the same ‘ProductEntity’ name: the interface and a constant. This is possible because a TypeScript declaration can create entities in two independent groups:
  + **Types**: this group is only visible by the TS compiler, and elements in this group will be used then defining the type of a variable or member. The interface ProductEntity belongs to this group.
  + **Values**: this group will be compiled down to JavaScript and the entities represent real JavaScript objects that can be used. The const ProductEntity belongs to this group.

Since the two names are in two different groups, there is no duplicated name error. This has the benefit of importing it once and being able to use both elements.

* **What does Type<ProductEntity>(“Product”) do?** This tiny data structure is used as a statically-typed way of providing type information.
  + On one side it contains the name of the entity in a string field, so it can be used at run-time to identify the type of the entity.
* On the other side is generic, containing the type information at compile-time to enable statically-type APIs, like for example the one to create PropertyRoute: ProductEntity.propertyRoute(p => p.name)

## Regions and Territories

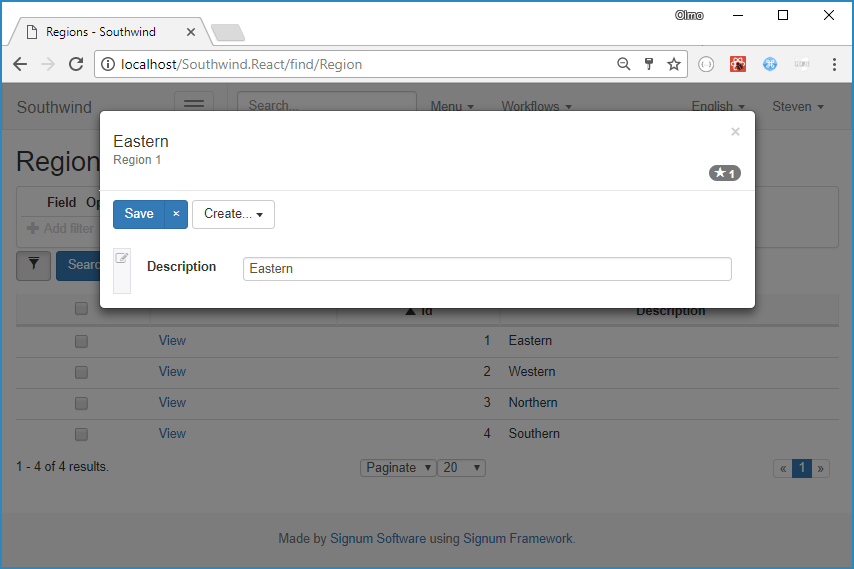
If we run the application and write “Regions” in the Omnibox (search box on the top) we see that we have a fully-featured search control.



This component is quite powerful: it uses the query registered in logic as the initial query and lets the user order, filter, add or remove columns, get into collections, etc. There are global and query-specific extension points to customize its behavior and appearance, but most of the time it covers all the features that your users need without having to write any single line of client-side code. What else you can ask for?.

At the end of this tutorial we will make use of some of this configuration options and extension points, but for now we will use the default behavior.

When we click view (or create [+]), we get into the component of the region itself.



Let’s see how it works:

If we go to SouthwindClient.tsx we see that it’s mainly a big start function registering views (Navigator.addSettings) and operations (Operations.addSettings).

We have views registered for most of the entities defined in previous tutorials but there are some missing (commented-out) ones: RegionEntity, ShipperEntity and TerritoryEntity.

### Dynamic Component

Where is this view implemented then? Nowhere. It’s automatically generated from the meta-data by DynamicComponent or DynamicViewComponent.

* **DynamicComponent**, defined in Signum.React, creates an automatic view for an entity with no customization options at all. Useful for simple entities.
* **DynamicViewComponent**, defined in Signum.React.Extensions, overrides DynamicComponent and allows end-users to customize the view by choosing the components, rearranging them and adding some interaction. The result is stored in the database in a custom JSON format but then is rendered using the same React components (EntityLine, ValueLine, etc…) than a custom react components for an entity. It’s able to express more complicated components than DynamicComponent, and is able to export the custom JSON format into valid Typescript / React code if necessary.

Thanks to these dynamic controls we can save ourselves from writing a lot of UIs for entities that are going to be rarely used, providing with no additional development effort a comprehensive UI for all the application entities, ultimately empowering our users to control any corner of the database.

On the other side, as with any other kind of automatic behavior, there is a loose in control. Often we need special behavior or layout/style that requires writing custom components for our entities. Let’s see how we can do this with other entities.

## Product (custom React component)

This time instead of letting DynamicComponent figure out UI for us, we will see how can be written by ourselves:

In SouthwindClient.tsx we will need to register an EntitySettings for Product entity

Navigator.addSettings(new EntitySettings(ProductEntity, p => import('./Templates/Product')));

Let’s analyze this code:

* **Navigator** is the main module for registering components, navigating to entities, opening entities in pop-ups or retrieving entities from the server.
* **addSettings** allows registering an EntitySettings for a particular Entity type.
* **EntitySettings** contains configuration options for a particular entity type. Typically the component to show but there are many other interesting options available.
* **ProductEntity** is an auto-generated constant of type Type<ProductEntity>, defined in Southwind.Entities.ts, is equivalent to writing “Product” but statically typed.
* **p => import(…)** this lambda allows us to delay the request of the component until it is used. `import` is a new feature in ES7 that webpack bundler uses for code-spliting. In this case we want to separate the main bundle (where Main.tsx and all the Client modules are) from all the entity react components that will be loaded on demand.

If we press F12 on the string inside of the import statement, we can see the code:

import \* as React from 'react'

import { ProductEntity } from '../Southwind.Entities'

import { ValueLine, EntityLine, EntityCombo, EntityList, EntityDetail, EntityStrip, EntityRepeater, TypeContext } from '../../../../Framework/Signum.React/Scripts/Lines'

export default class Product extends React.Component<{ ctx: TypeContext<ProductEntity> }> {

    render() {

        const ctx = this.props.ctx;

        return (

            <div>

                <ValueLine ctx={ctx.subCtx(p => p.productName)} />

                <EntityLine ctx={ctx.subCtx(p => p.supplier)} />

                <EntityLine ctx={ctx.subCtx(p => p.category)} />

                <ValueLine ctx={ctx.subCtx(p => p.quantityPerUnit)} />

                <ValueLine ctx={ctx.subCtx(p => p.unitPrice)} />

                <ValueLine ctx={ctx.subCtx(p => p.unitsInStock)} />

                <ValueLine ctx={ctx.subCtx(p => p.reorderLevel)} />

                <ValueLine ctx={ctx.subCtx(p => p.discontinued)} />

            </div>

        );

    }

}

Let’s analyze this code:

* First we have the import statements:
  + We include everything in **‘react’ NPM package** under the local variable ‘React’
  + We include only the interface/constant ProductEntity from the auto-generated file one folder up ‘../Southwind.Entities.ts’.
  + We include a battery of property components (ValueLine, EntityLine, etc…) from Lines.ts defined in framework.
* After that we define our component. Some considerations:
  + We export this class as the ‘export default’ making this class the main exported element in this JavaScript module (file).
  + Our component is a simple class that inherits from ‘React.Component<P, S>’
  + The **P** is the generic argument used to define the public properties of the react component, and in this case is an in-line declared interface of an object containing only one property, ‘ctx’, of type TypeContext<ProductEntity>.
  + The component has no state (**S**).
* Inside of our component, we have just one `render` function.
  + The function returns only one element, a div with many children.
  + Each property contains one component, in this case an EntityLine for reference to other entities, and ValueLine for simple values.
  + Each component knows to what property is *bound* by using `ctx.subCtx(p => p.productName)`.

If you have some experience with react components some code should look familiar, but you may be surprised by the simplicity of the code.

* There are no callbacks from the events to set the values back to the properties, as is usual in React unidirectional data flow.
* There is no code for decorating components with error messages.
* There is no reference to bootstrap rows and columns or any other css information.
* There are no localizable strings.
* There are no conditional visibility depending on the role.

TypeContext, inheriting from StyleContext, is the piece of magic that makes all this possible. Let’s see how it works:

#### StyleContext

StyleContext contains only two members:

* A **parent** **StyleContext** that will be used as a fallback when the options are not defined in the child style options.
* A **StyleOptions**, with style information that is typically shared between different components in the same page to preserve consistency, like:
  + **formGroupStyle**: Determines the way the FormControl of each property component should be rendered, based on the alternatives in <https://getbootstrap.com/docs/3.3/css/#forms>. Contains the following options:
    - **None**: Only the value is rendered.
    - **Basic**: Label on top, value below.
    - **BasicDown**: Value on top, label below.
    - **SrOnly**: Label is not visible except for Screen-Readers
    - **LabelColumns**: Default value, label on the left, value on the right (except on RTL scenarios).
  + **formGroupSize**: Determines the size of the property components. Contains the following values:
    - **Normal**: Bootstrap default
    - **Small**: Signum.React default
    - **ExtraSmall**: Like in the SearchControl filters.
  + **placeholderLabels:** For components that allow placeholders, like a textbox, writes a placeholder with the name of the property.
  + **readOnly:** Determines if the components should be read-only or not.
  + **formControlClassReadonly:** Customizes how read-only components should be seen. Typically ‘form-control-static’ (without border) or ‘form-control readonly’ (with border).
  + **labelColumns:** Determines how many bootstrap columns should the label take. Only usedwhen formGroupStyle=”LabelColumns”**.**
  + **valueColumns:** Determines how many bootstrap columns should the value take. Only usedwhen formGroupStyle=”LabelColumns”**.** By default is (12 – labelColumns).
  + **frame:** Set on the root StyleOptions, keeps a reference to the top-most entity container, enabling certain scenarios.

#### TypeContext

This class inherits from StyleContext and adds two important pieces of information:

* **Binding<T>**: Represents a read/write access to a particular property in a parent object. Thanks to this object, we don’t need to create a callback for each component to set the value back into the entity.
* **PropertyRoute:** Represents a piece of metadata related to an Entity, Property, or a sequence of Properties until a database column is unambiguously determined (in case of embedded entities).This metadata provides a lot of information for each property, like:
  + **Type and Implementations:** useful for example to determine the right component for a ValueLine (textbox/date-time picker/combo box). Also the implementations in case of polymorphic foreign keys is available.
  + **Format:** How a number or a date should be formatted, number of decimals, date or date-time, etc…
  + **Unit:** Unit of a numeric number ($, €, mins, etc…)
  + **Security:** Whether the current user is allowed to see one particular property or should be hidden from the UI. (The values won’t be serialized anyway, but the framework needs to know that the component should not be visible)
  + **Localization:** Name of the property in the current user language.
  + **Other information:** Max string length, string is MultiLine, etc…

Aditionally the TypeContext provides some convenience properties:

* **value**: To get/set the current value in the binding.
* **error**: To get/set the error information for the binding property. Useful for validation.

By combining three pieces of information (Style, Binding and Metadata), TypeContext becomes the only thing you need to provide to a property component in order to get a consistent look-and-feel and excellent default behavior with one line of code.

Additionally, TypeContext<T> is generic so you get all the benefits of statically-typed code.

##### Creating a TypeContext

There are three ways of creating a TypeContext<T>:

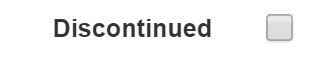
* **TypeContext.root(someEntityInstance):** This static method creates a new TypeContext for an entity instance:
  + The style information comes from an optional StyleOptions and an optional parent StyleContext.
  + The Metadata comes from the run-time type of the entity instance.
  + The binding is a read-only binding to get access to the entity itself.
* **TypeContext.subCtx(lambda):** This instance method creates a sub-context from a previous context passing a statically-typed lambda expression accessing a property. This lambda won’t be directly evaluated, instead it will be converted to string and parsed, to get the name of the property but keep the API statically-typed.
  + The style information comes from an optional StyleContext and the parent TypeContext.
  + The Metadata comes from accessing the property information of the parent metadata for the current property name.
  + The binding will be made using the parent value (using the parent binding) and the property name.
* **TypeContext.subCtx(MixinType):** In order to access Mixin in entities, a very similar overload is provided that instead of taking a lambda uses a Mixin type constant. (Note: getMixin is no supported in the lambda itself!)

#### Types of Property Components

Creating the Product component we have been introduced to two common property components: EntityLine and ValueLine. Let’s see how they differentiate and what other alternatives exist:

##### ValueLine

Represents a property of a simple value, internally determines the right component for the property using the PropertyRoute.

* + **CheckBox**: Default for non-nullable booleans.
  + **ComboBox**: Default for enums and nullable booleans.

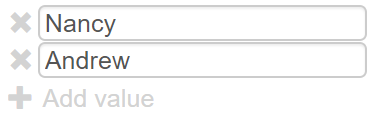


* + **DateTime**: Default for dates, a date-time picker sensible to the FormatAttribute (or DateTimeValidation) to detect if the time part is necessary.



* + **Textbox**: Default for strings.
  + **TextArea**: Default for multi-line strings (StringLengthValidator(MultiLine= true)).
  + **Number**: Default for numbers without decimal places. A text box that prevents letters. Sensible to format and unit.
  + **Decimal**: Default for numbers with decimal places. A text box that prevents letters. Sensible to format and unit.   
    
  + **TimeSpan**: Default for TimeSpan, a text box that only allows valid time spans. Sensible to format.

###### MultiValueLine

Designed to represent a MList<T> when T is some simple value. Similar to what the search control uses when using IsIn / IsNotIn. Is small enough to share his space with a Label.   


###### EnumCheckboxList

Useful to represent a MList<SomeEnum> as a list of checkboxes. Similar to EntityCheckboxList.

##### EntityBase

Abstract base class that represents a relationship to another entity. Works seamlessly with Entities and Lites. Allowing for important actions:

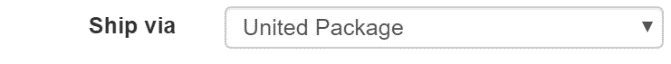
* **Create:** When the property is null, opens a modal window to create a new entity instance and store it in the property.
* **Find:** When the property is null, opens a modal SearchControl to find an already-existing instance in the database and set it in the property.
* **View:** When the property is set, opens the related entity in a modal window.
* **Remove:** Removes the entity from the property, destroying the relationship (but not deleting the entity from the database).

###### EntityLine

Inherits from EntityBase and adds autocomplete functionality. Useful for relationships with independent entities.

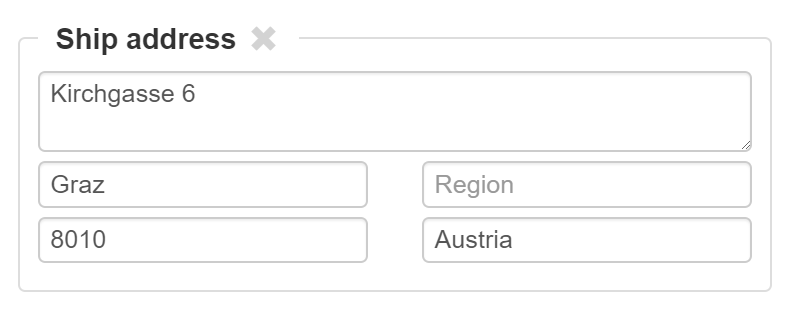


###### EntityCombo

Inherits from EntityBase and adds a combobox functionality that is automatically loaded with all the entities in the related table, in alphabetical order. Useful for relationships with independent entities in tables with few rows (IsLowPopuplation=true)

###### EntityDetail

Inherits from EntityBase and renders the related entity in-place. Useful for EmbeddedEntity or Entities that are Part or SharedPart.

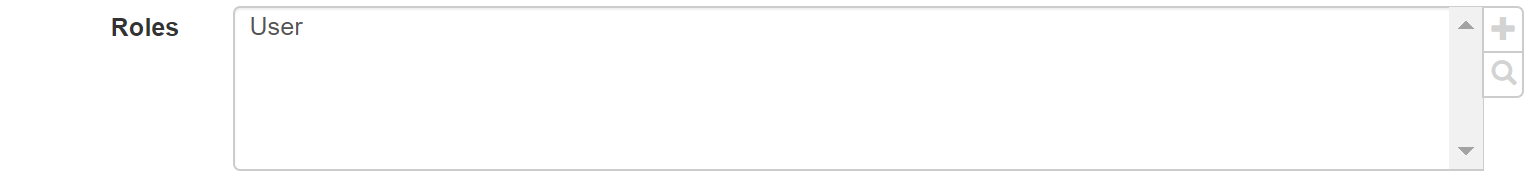


##### EntityListBase

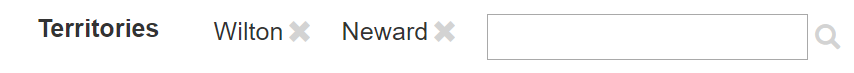
Abstract base class that represents a relationship to many entities in a MList<Entity> or MList<Lite<Entity>>.

* **Create:** Opens a modal window to create a new entity instance and add it to the list.
* **Find:** Opens a modal SearchControl to find an already-existing instances in the database and add them to the list.
* **View:** Opens the selected entity in the list in a modal window.
* **Remove:** Removes the selected entities from the list, destroying the relationship (but not deleting the entity from the database).
* **Move:** Allows reordering entities in a list.

###### EntityList

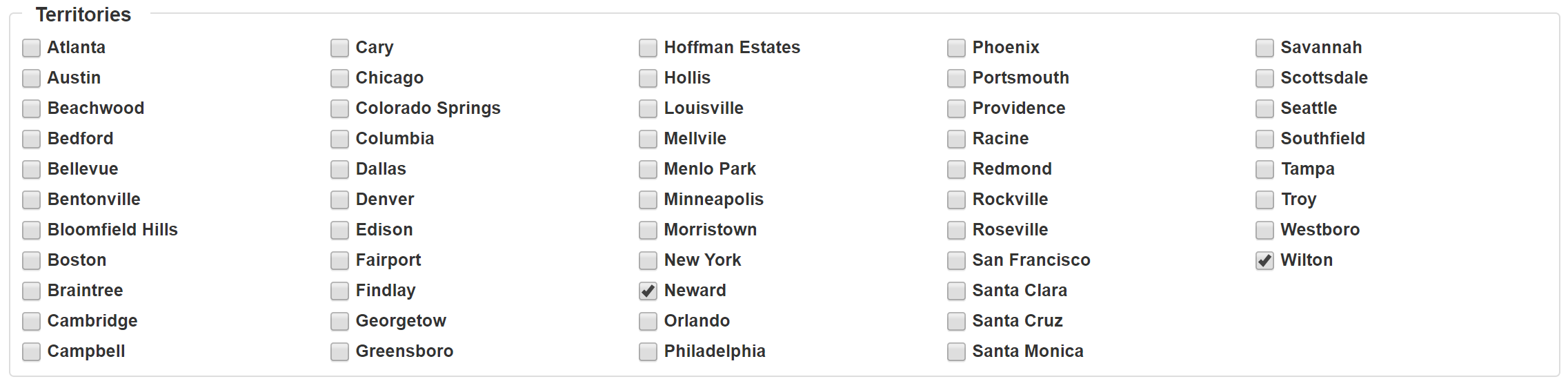
Inherits from EntityListBase and shows the elements in a selectable list. (Select multiple). Rarely used nowadays.

###### EntityStrip

Inherits from EntityListBase and shows the elements in a strip of tag-like pills. They can be independently removed and added using auto-completion.

###### EntityCheckboxList

Inherits from EntityListBase and shows the elements as a list of checkboxes in alphabetical order and arranged in columns. Useful for relationships with independent entities in tables with few rows (IsLowPopuplation=true).

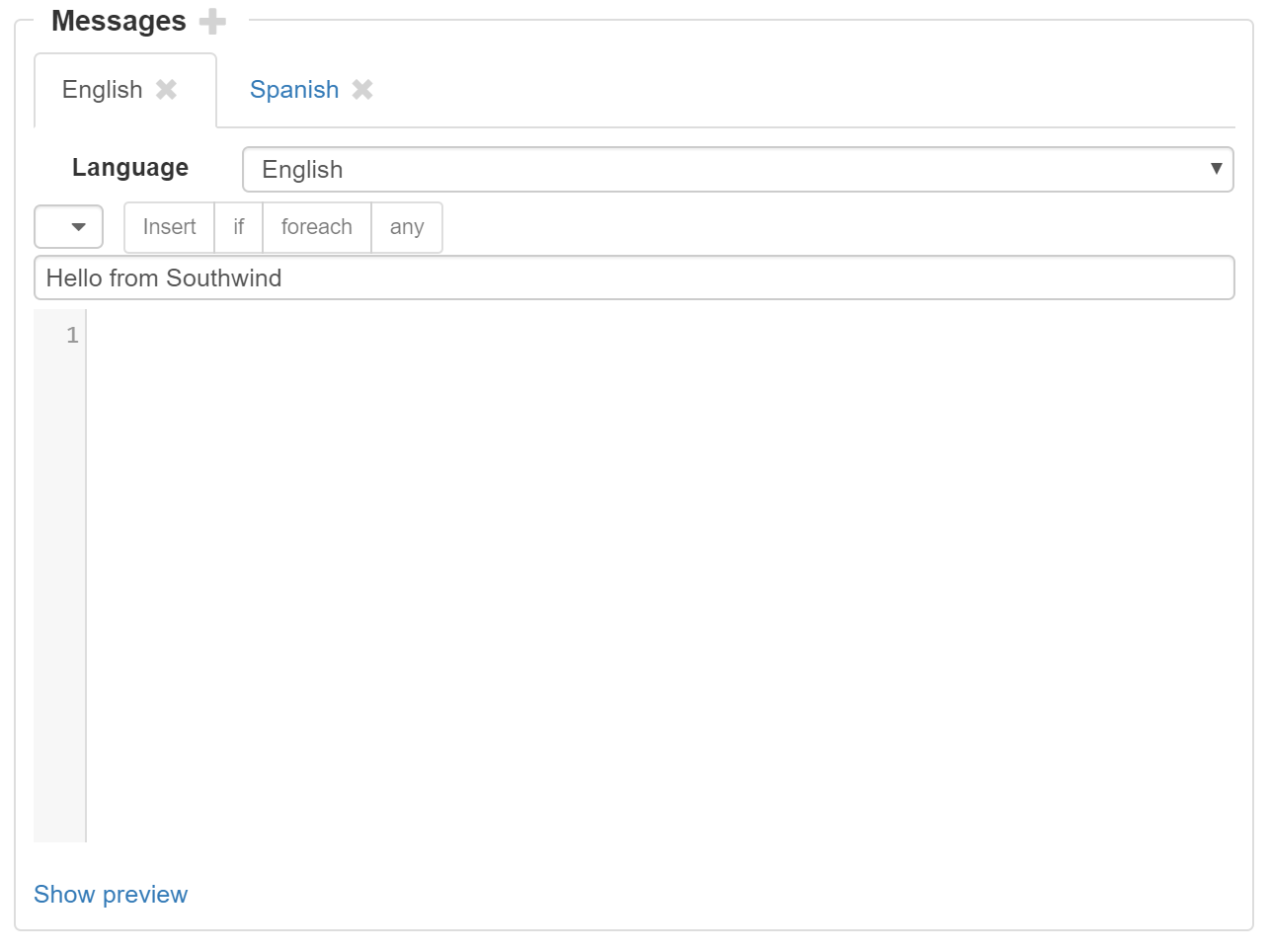


###### EntityRepater

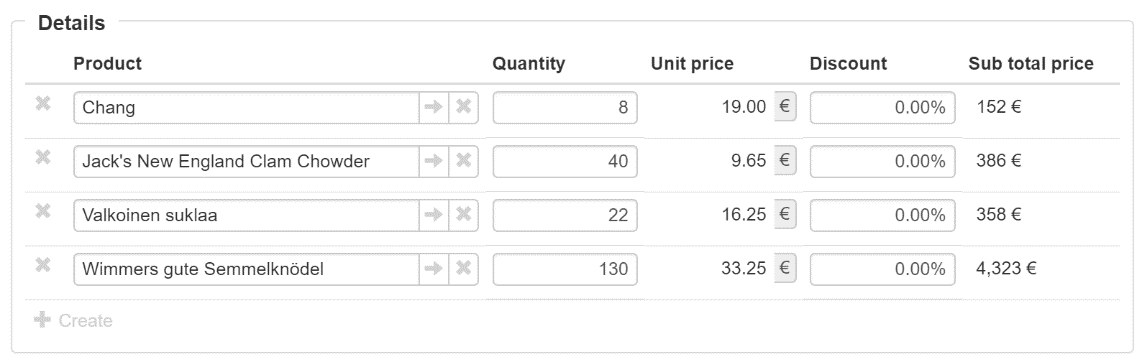
Inherits from EntityListBase and renders the related entities in-place, one after the other. Useful for lists of embedded entities or part entities.   


###### EntityTabRepater

Inherits from EntityListBase and renders the related entities in-place, in different tabs. Useful for lists of EmbeddedEntits or Entities that are Part or SharedPart.



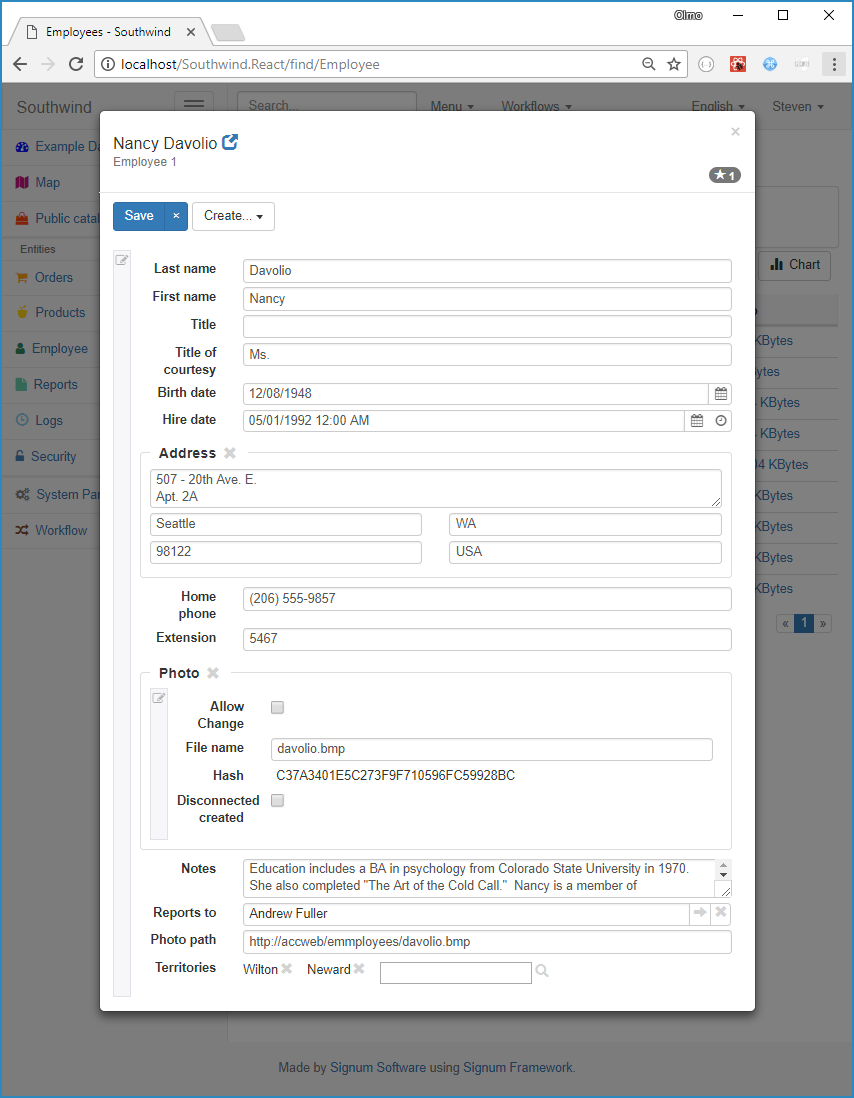
###### EntityTable

Inherits from EntityListBase and renders the related entities in a table. Useful for lists of EmbeddedEntits or Entities that are Part or SharedPart.

## Employees (using State)

Now that we know how TypeContext works, and how to build a custom react component, let’s try to do something more difficult.

Let’s take a look at what DynamicViewComponent can do for Employees:



The automatically generated component has chosen one component for each property in the order they are defined in the entity. The selection is quite reasonable, textboxes for properties, date-time pickers for dates, and even more advanced components for selecting entities or list of entities.

There are some clear problems, like the photo of the employee not being shown, and maybe we could layout the components differently. Let’s see how to create a custom component.

There are two ways of creating a custom component:

* You could use **ReactCodeGenerator** to make at design-time what the **DynamicComponent** does at run-time. This approach is good if you need to generate many entities at once, with many properties each, but requires a little bit of initial setup, and switching to the Load application.
* You could create it manually, it’s not too much code and very often the fastest approach. Also we’re here to learn.

Instead of placing the properties one after the other, we’ll start grouping them in logical blocks, using fieldset and legend.

export default class Employee extends React.Component<{ ctx: TypeContext<EmployeeEntity> }> {

    render() {

        const ctx = this.props.ctx;

        return (

            <div>

                <fieldset>

                    <legend>Personal Info</legend>

                    <ValueLine ctx={ctx.subCtx(p => p.title)} />

                    <ValueLine ctx={ctx.subCtx(p => p.firstName)} />

                    <ValueLine ctx={ctx.subCtx(p => p.lastName)} />

                    <ValueLine ctx={ctx.subCtx(p => p.birthDate)} />

                    <ValueLine ctx={ctx.subCtx(p => p.homePhone)} />

                </fieldset>

                <EntityDetail ctx={ctx.subCtx(p => p.address)} />

                <fieldset>

                    <legend>Company data</legend>

                    <ValueLine ctx={ctx.subCtx(e => e.titleOfCourtesy)} />

                    <EntityLine ctx={ctx.subCtx(e => e.reportsTo)} />

                    <ValueLine ctx={ctx.subCtx(e => e.hireDate)} />

                    <ValueLine ctx={ctx.subCtx(e => e.extension)} />

                    <EntityStrip ctx={ctx.subCtx(e => e.territories)} />

                </fieldset>

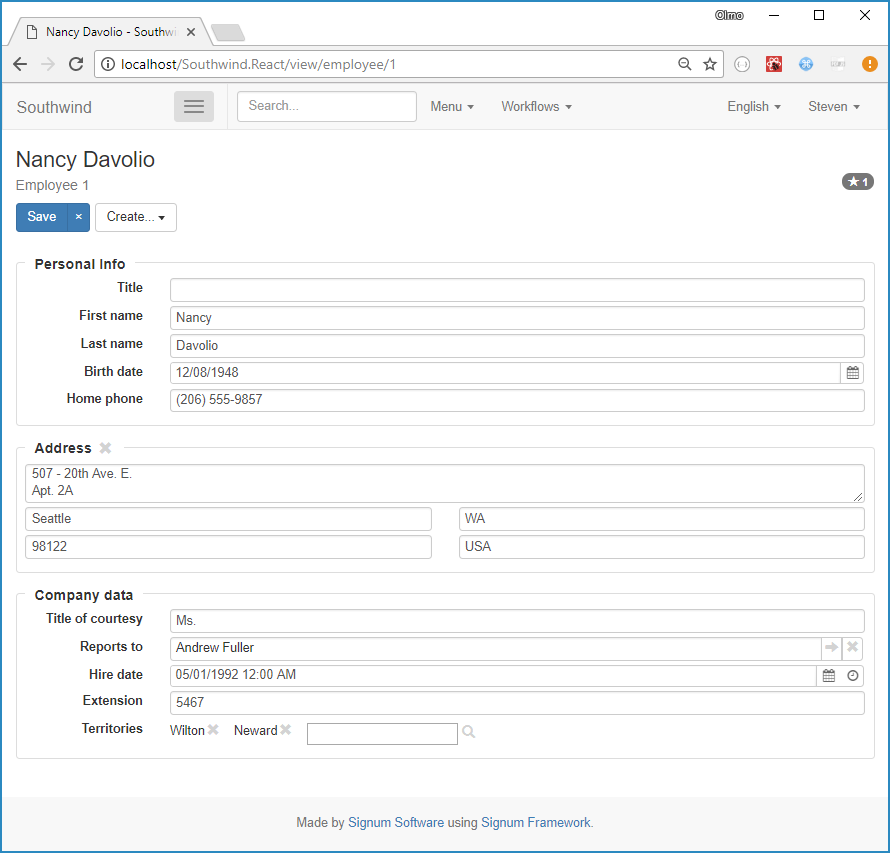
            </div>

        );

    }

}

This is the result:



We could also place Title, FirstName and LastName in the same row, and use placeholders to show the titles. Let’s see how:

export default class Employee extends React.Component<{ ctx: TypeContext<EmployeeEntity> }> {

    render() {

        const ctx = this.props.ctx;

        const ctxBasic = ctx.subCtx({ formGroupStyle: "SrOnly" });

        return (

            <div>

                <fieldset>

                    <legend>Personal Info</legend>

                    <div className="form-vertical row">

                        <div className="col-sm-2">

                            <ValueLine ctx={ctxBasic.subCtx(p => p.title)} placeholderLabels={true} />

                        </div>

                        <div className="col-sm-5">

                            <ValueLine ctx={ctxBasic.subCtx(p => p.firstName)} placeholderLabels={true} />

                        </div>

                        <div className="col-sm-5">

                            <ValueLine ctx={ctxBasic.subCtx(p => p.lastName)} placeholderLabels={true} />

                        </div>

                    </div>

                    <ValueLine ctx={ctx.subCtx(p => p.birthDate)} />

                    <ValueLine ctx={ctx.subCtx(p => p.homePhone)} />

                </fieldset>

…

            </div>

        );

    }

}



Finally, will be nice to see the photo of the employee (even if it’s a low-resolution photo from the 90s…). In order to do this, we’ll need to make a small break to explain two important concepts from React: props and state.

### Props

Many React components have only props that control his appearance and behavior from the parent component. In Signum for example, most of your component receive a TypeContext from the parent component (FramePopup or FramePage).

Props are read-only, meaning that you cannot change the value stored in some prop for another one, but just as arguments of a function, nothing prevents you to modify the properties of the object in case of being a mutable object by reference, like the entity in a TypeContext.

Note: This is not considered the purest React approach, instead some tutorials encourage to use Immutable.js or frameworks like Redux/Flux, etc... Signum chooses a more practical and direct way for changing the entities.

These pure components typically are not able to modify any state by themselves, instead they call a callback function to ask the parent to do it for them.

#### Example:

In a **DateTimePicker** component we have two props:

* **selectedDate:** Of type Date, lets the parent control the date that is selected.
* **onSelectedDateChange:** Callback that the DateTimePicker will use to notify the parent that the user has clicked on a particular date of the calendar, of written a valid date format. The parent component will then update their state and re-render (using setState or forceUpdate).

### State

State gives components some independence from their parents by letting them have some mutable state that is kept private. This state can be completely independent from the props, but very often is related and has to be kept synchronized if the props happened to change. For example, retrieving information from the server depending on props data.

#### Example:

Again in a **DateTimePicker** component, we have two pieces of state:

* **isOpen:** For simplicity, the parent component of the DateTimePicker is not interested in whether the calendar popup is open or not. This information is only interesting for the DateTimePicker, and can be kept private.
* **currentDate:** Once open, the user is free to navigate between the months in the calendar without actually selecting a date yet. This is also information that is needed to render and control the behavior, but is superfluous for the parent component.

In our particular case, the Employee component will need a new piece of information: the FileEntity with the photo. The Employee only contains a Lite<FileEntity> that will need to be retrieved from the database.

Let’s start by declaring the shape of the state as the second generic argument of React.Component<P, S>:

export default class Employee extends React.Component<{ ctx: TypeContext<EmployeeEntity> }, { photo?: FileEntity }> {…}

This long line could be confusing at first, because we are declaring the types in-line. This is an equivalent declaration:

interface EmployeeProps {

    ctx: TypeContext<EmployeeEntity>;

}

interface EmployeeState {

    photo?: FileEntity;

}

export default class Employee extends React.Component<EmployeeProps, EmployeeState> {…}

Once you have state, you need to define it in the constructor of the component:

constructor(props: EmployeeProps) {

    super(props);

    this.state = { photo: undefined };

}

As you see, the photo field in the state is optional, and initialized to undefined when the component is created. If will take some time for the request of the server to retrieve the photo.

Before the component is mounted in the DOM, we will get called to componentWillMount. This is a great time to fire our request for the photo:

componentWillMount() {

    this.loadPhoto(this.props);

}

componentWillReceiveProps(p: EmployeeProps) {

    this.loadPhoto(p);

}

loadPhoto(props: EmployeeProps) {

    var e = props.ctx.value;

    if (is(e.photo, this.state.photo))

        return;

    if (!e.photo)

        this.setState({ photo: undefined });

    else if (e.photo.entity)

        this.setState({ photo: e.photo.entity });

    else

        this.setState({ photo: undefined }, () => {

            Navigator.API.fetchAndForget(e.photo!)

                .then(ph => this.setState({ photo: ph }))

                .done();

        });

}

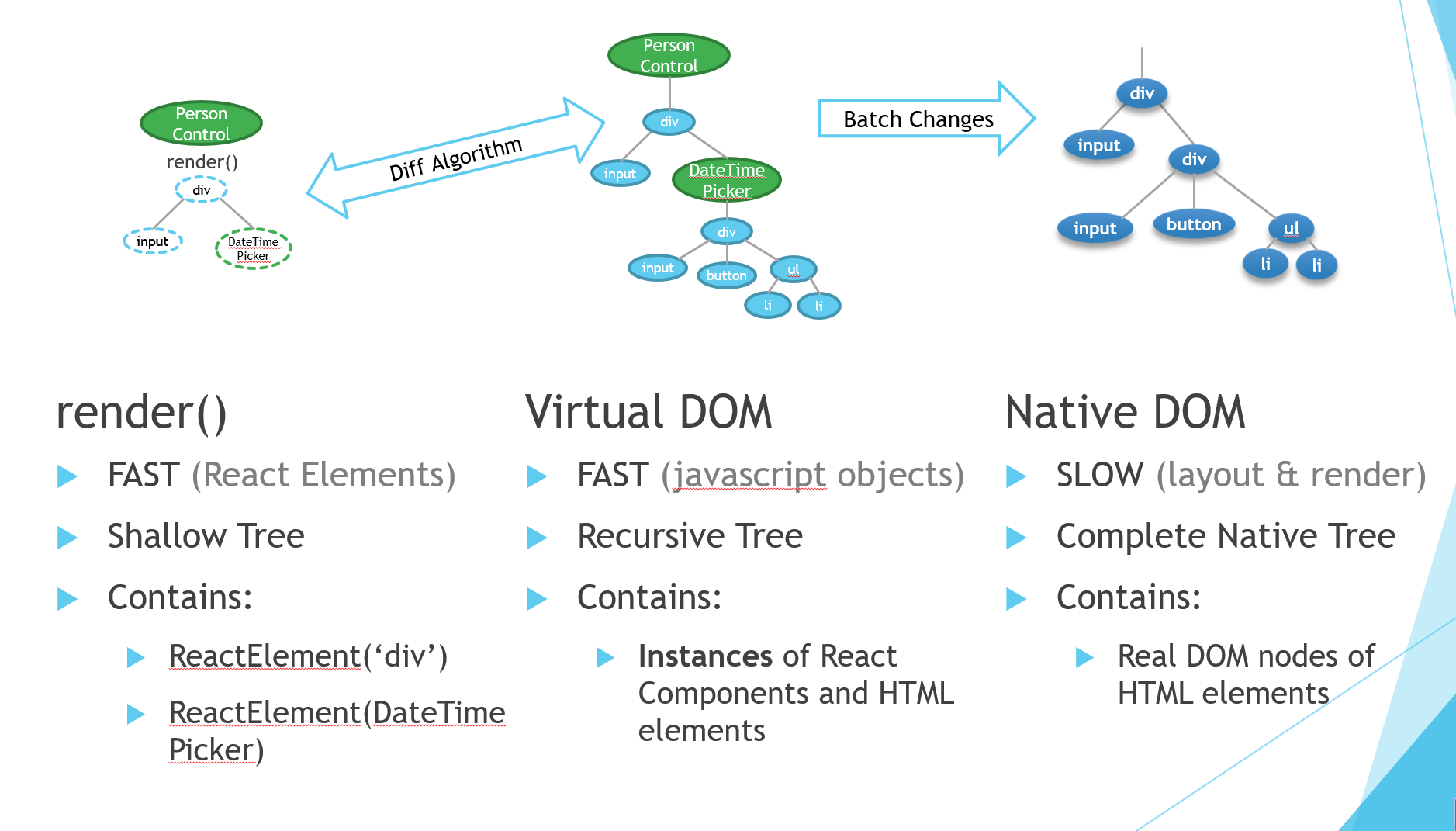
Let’s analyze the code:

* First we compare if the photo we have in the state is the same that the employee is pointing to, using Signum ‘**is’** function, that is able to compare Entity and Lite <Entity> seamlessly. In this case we return.
* Otherwise, we check if the employee has no photo, then we call setState to remove the photo from the state. The change will be queued and, at some point, react will apply it and re-render the view.
* If the employee has a photo, but is already loaded (like when the photo is new), we don’t need to retrieve it, we can directly store it the state.
* Finally if we need to retrieve it from the server, we temporally set the photo field to undefined and, when the change is applied in the UI (async but very fast), we trigger the request using a generic API already provided by the framework to retrieve an Entity from a Lite<Entity>.
  + Using promises we set a callback that will be called when the entity is retrieved, and setting the state and eventually re-rendering.
  + If the server is unreachable, or any other exception happens, the promise will be set as faulted. You can manually catch the exception using **catch** but typically you can use Signum **done** function to capture any exception and show an error message to the UI. Done should be used on any promise that is not returned.

Also, check how loadPhoto is not only called from componentWillMount, also by componentWillReceiveProps. In order to understand this, we need to make again a break to understand another important concept of React, the VirtualDOM:

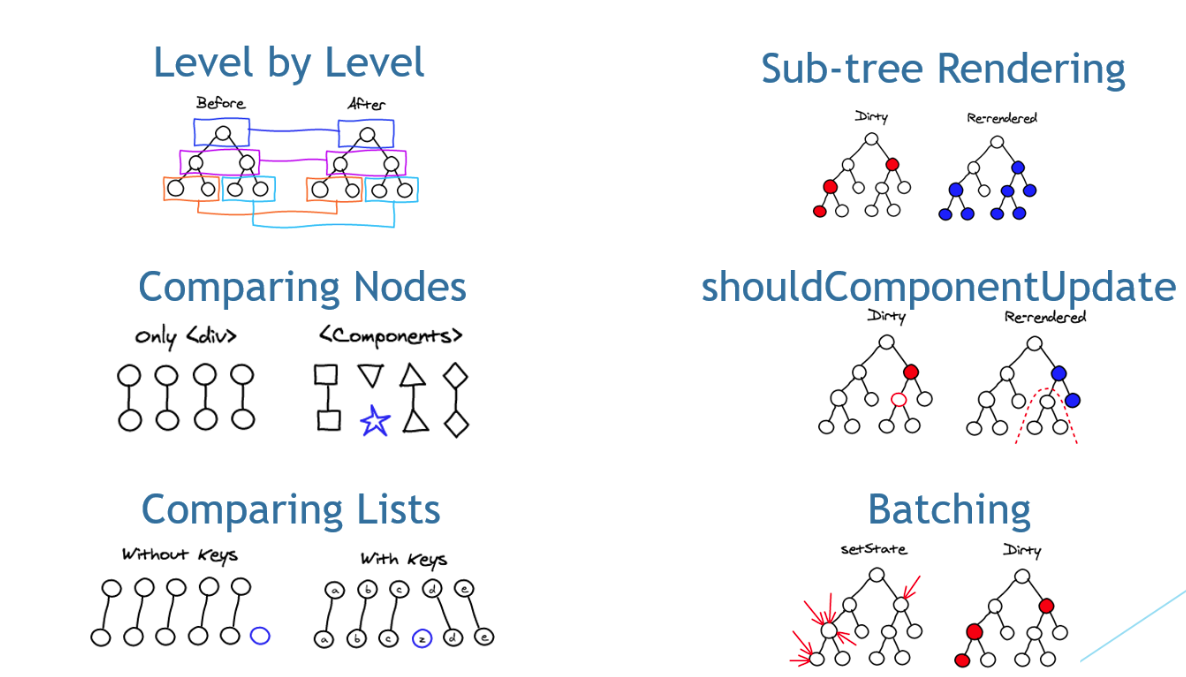
### VirtualDOM

When you write JSX in your render function, your HTML is not placed directly in the native DOM of the browser, this will create flickering, losing the cursor, and bad performance. Instead, React keeps an internal data structure called VirtualDOM that represents what react thinks should be in the screen at any given time. The VirtualDOM has some advantages over the native DOM.

* Since it contains just JavaScript objects, it’s quite fast to read and modify.
* It contains a mixture of native (intrinsic) HTML nodes together with react components, containing their props and state.
* Changes made on the VirtualDOM can be batched to the real DOM, improving performance.

At any given execution of a component render function, the result is compared with the VirtualDOM using a diff algorithm, and the necessary components are created, removed, or updated.

The algorithm is fast and simple, it doesn’t try to move nodes up or down in the hierarchy, reusing only the nodes that are in the same position in the tree and have the same type.



What could happened, however, is that a component that started with some property values, gets re-rendered with other properties. If the component has only props this is typically not a problem, but as long as there is state that depends on this props, the state will need to be recalculated. We can do this by intercepting the call to componentWillReceiveProps:

componentWillReceiveProps(p: EmployeeProps) {

    this.loadPhoto(p);

}

Finally we need to extend our render function with the photo.

render() {

        const ctx = this.props.ctx;

        const ctxBasic = ctx.subCtx({ formGroupStyle: "SrOnly" });

        return (

            <div className="row">

                <div className="col-sm-9">

                    <fieldset>

                        …

                    </fieldset>

                    <EntityDetail ctx={ctx.subCtx(p => p.address)} />

                    <fieldset>

…

                    </fieldset>

                </div>

                <div className="col-sm-3">

                    {/\*photo\*/}

                    <FileLine ctx={ctx.subCtx(e => e.photo)} onChange={() => this.loadPhoto(this.props)} />

                    {this.state.photo && <img className="img-responsive" src={"data:image/jpeg;base64," + this.state.photo.binaryFile} />}

                    {/\*photo\*/}

                    <div className="form-vertical">

                        <ValueLine ctx={ctx.subCtx(e => e.notes, { formGroupStyle: "Basic" })} valueHtmlAttributes={{ rows: 10, className: "notes" }} />

                    </div>

                </div>

            </div>

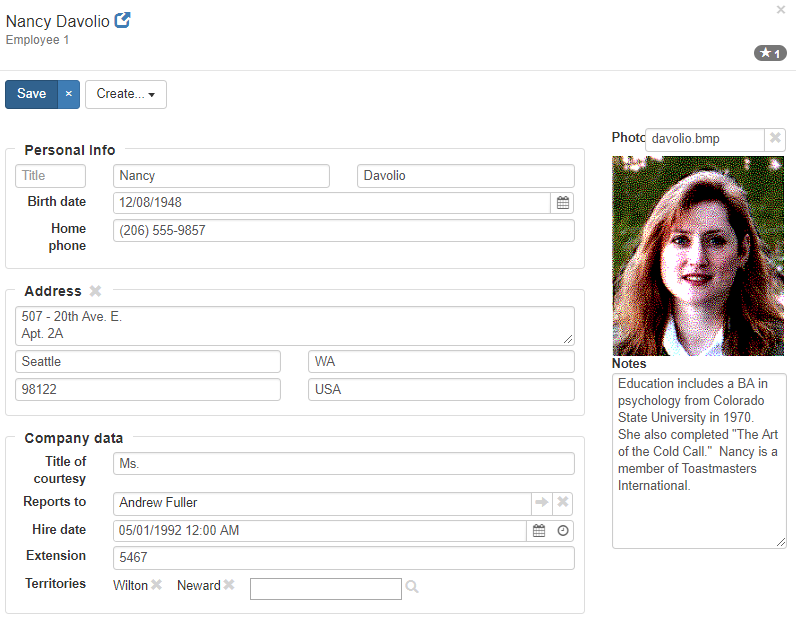
        );

    }

We use a FileLine from Signum.Extensions to upload and download the Lite<FileEntity> for the photo.

Also, we render the photo below using a data-url. Thanks to [TypeScript non-nullable types](https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-0.html), the compiler will help us aware of the case where the photo is undefined because is not yet loaded.

Finally, we use the remaining space to place a TextArea with the notes for the employee. Let’s see the end result.



As you have seen, controlling state is a big source of complexity in a React.Component. It is typically a good idea to factor out the parts that require state (in this case the photo) in another component. Let’s create then the EmployeePhoto component using reactClassState snippet.

The final code looks like this:

export default class Employee extends React.Component<{ ctx: TypeContext<EmployeeEntity> }>

{

    render() {

        const ctx = this.props.ctx;

        const ctxBasic = ctx.subCtx({ formGroupStyle: "SrOnly" });

        return (

            <div className="row">

                <div className="col-sm-9">

                    <fieldset>

                        <legend>Personal Info</legend>

                        <div className="form-vertical row">

                            <div className="col-sm-2">

                                <ValueLine ctx={ctxBasic.subCtx(p => p.title)} placeholderLabels={true} />

                            </div>

                            <div className="col-sm-5">

                                <ValueLine ctx={ctxBasic.subCtx(p => p.firstName)} placeholderLabels={true} />

                            </div>

                            <div className="col-sm-5">

                                <ValueLine ctx={ctxBasic.subCtx(p => p.lastName)} placeholderLabels={true} />

                            </div>

                        </div>

                        <ValueLine ctx={ctx.subCtx(p => p.birthDate)} />

                        <ValueLine ctx={ctx.subCtx(p => p.homePhone)} />

                    </fieldset>

                    <EntityDetail ctx={ctx.subCtx(p => p.address)} />

                    <fieldset>

                        <legend>Company data</legend>

                        <ValueLine ctx={ctx.subCtx(e => e.titleOfCourtesy)} />

                        <EntityLine ctx={ctx.subCtx(e => e.reportsTo)} />

                        <ValueLine ctx={ctx.subCtx(e => e.hireDate)} />

                        <ValueLine ctx={ctx.subCtx(e => e.extension)} />

                        <EntityStrip ctx={ctx.subCtx(e => e.territories)} />

                    </fieldset>

                </div>

                <div className="col-sm-3">

                    {/\*photo\*/}

                    <FileLine ctx={ctx.subCtx(e => e.photo)} onChange={() => this.forceUpdate()} />

                    <EmployeePhoto lite={ctx.value.photo} />

                    {/\*photo\*/}

                    <div className="form-vertical">

                        <ValueLine ctx={ctx.subCtx(e => e.notes, { formGroupStyle: "Basic" })} valueHtmlAttributes={{ rows: 10, className: "notes" }} />

                    </div>

                </div>

            </div>

        );

    }

}

interface EmployeePhotoProps {

    lite: Lite<FileEntity> | null | undefined;

}

interface EmployeePhotoState {

    file?: FileEntity;

}

export class EmployeePhoto extends React.Component<EmployeePhotoProps, EmployeePhotoState> {

    constructor(props: EmployeePhotoProps) {

        super(props);

        this.state = { file: undefined };

    }

    componentWillMount() {

        this.loadPhoto(this.props);

    }

    componentWillReceiveProps(p: EmployeePhotoProps) {

        if (!is(p.lite, this.state.file))

            this.loadPhoto(p);

    }

    loadPhoto(props: EmployeePhotoProps) {

        var lite = props.lite;

        if (!lite)

            this.setState({ file: undefined });

        else if (lite.entity)

            this.setState({ file: lite.entity });

        else

            this.setState({ file: undefined }, () => {

                Navigator.API.fetchAndForget(lite!)

                    .then(file => this.setState({ file }))

                    .done();

            });

    }

    render() {

        if (!this.state.file)

            return null;

        return (<img className="img-responsive" src={"data:image/jpeg;base64," + this.state.file.binaryFile} />);

    }

}

## Orders

Our next step is now the Order entity. The central entity in Southwind database. This entity represents an order in the Southwind shop. It references to the Employee handling the order, the Customer that requested it and the address.

This is the initial, simplified UI:

export default class Order extends React.Component<{ ctx: TypeContext<OrderEntity> }> {

    render() {

        const ctx2 = this.props.ctx.subCtx({ labelColumns: { sm: 2 } });

        const ctx4 = this.props.ctx.subCtx({ labelColumns: { sm: 4 } });

        var o = ctx4.value;

        return (

            <div>

                <div className="row">

                    <div className="col-sm-6">

                        <EntityLine ctx={ctx2.subCtx(o => o.customer)} />

                        <EntityDetail ctx={ctx2.subCtx(o => o.shipAddress)} />

                    </div>

                    <div className="col-sm-6">

                        <ValueLine ctx={ctx4.subCtx(o => o.shipName)} />

                        {ctx2.value.isLegacy && < ValueLine ctx={ctx4.subCtx(o => o.isLegacy)} readOnly={true} />}

                        <ValueLine ctx={ctx4.subCtx(o => o.state)} readOnly={true}/>

                        <ValueLine ctx={ctx4.subCtx(o => o.orderDate)} readOnly={true} />

                        <ValueLine ctx={ctx4.subCtx(o => o.requiredDate)} />

                        <ValueLine ctx={ctx4.subCtx(o => o.shippedDate)} hideIfNull={true} readOnly={true} />

                        <ValueLine ctx={ctx4.subCtx(o => o.cancelationDate)} hideIfNull={true} readOnly={true} />

                        <EntityCombo ctx={ctx4.subCtx(o => o.shipVia)} />

                    </div>

                </div>

                <div className="row">

                    <div className="col-sm-4">

                        <EntityLine ctx={ctx4.subCtx(o => o.employee)} />

                    </div>

                    <div className="col-sm-4">

                        <ValueLine ctx={ctx4.subCtx(o => o.freight)} />

                    </div>

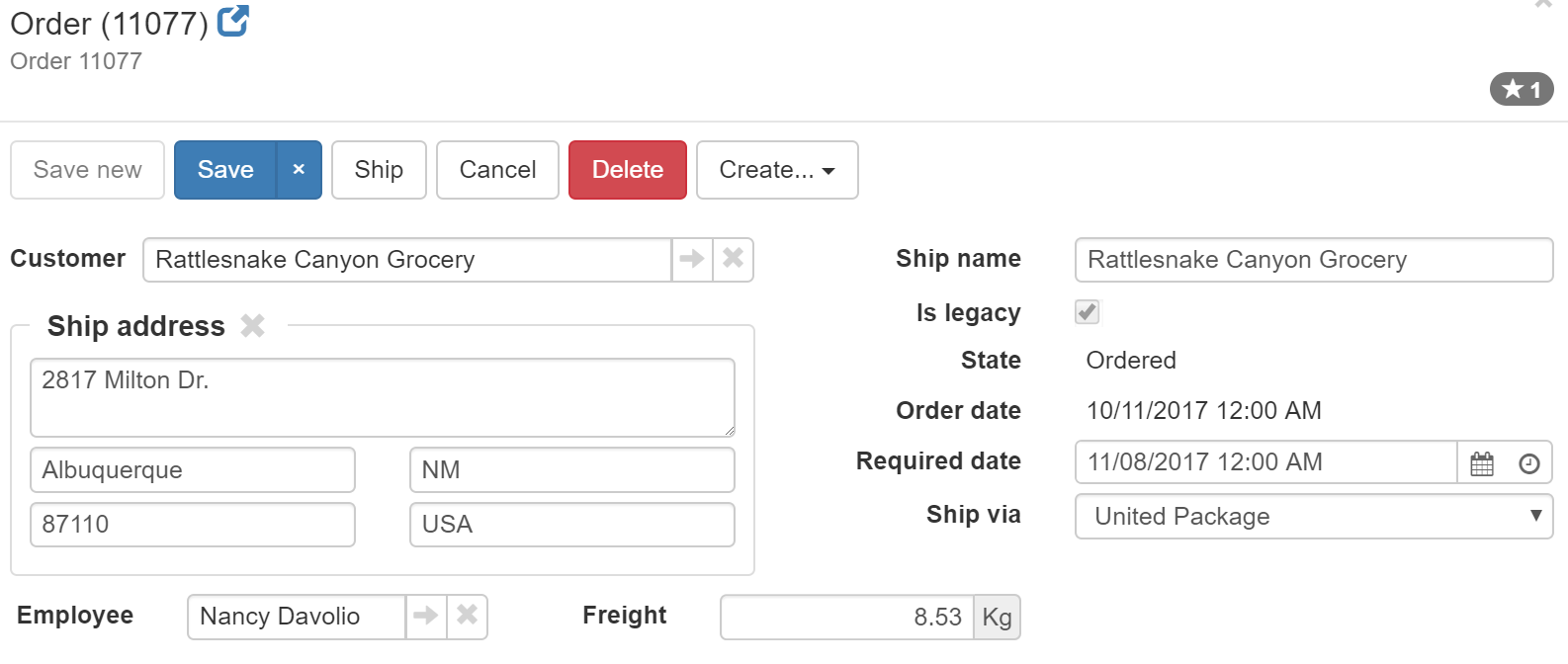
                </div>

            </div>

        );

    }

}



### Controlling Client-Side behavior:

One initial improvement that the users will love, is setting the Ship Address automatically when the Customer is set. We can do this by controlling onChange on the EntityLine for customer:

handleCustomerChange = (c: ChangeEvent) => {

    var order = this.props.ctx.value;

    var customer = c.newValue as CustomerEntity; //order.customer will also work

    order.shipAddress = customer == undefined ? undefined : { ...customer.address } as AddressEmbedded;

    order.modified = true;

    this.forceUpdate();

}

<EntityLine ctx={ctx2.subCtx(o => o.customer)} onChange={this.handleCustomerChange} />

What we’re doing is just getting the order from the TypeContext value, and setting the shipAddress.

Since we don’t want to have references to the same AddressEmbedded, we make a shallow clone using rest operator.

Finally we set the order as modified, necessary to avoid concurrency problems, and we forceUpdate to force a re-render with the new address.

### Customizing rendering of ValueLines

Other improvement will be to give a human-friendly description of the dates by making it relative to today. We can do this using fromNow from moment.js library and unitText property.

<ValueLine ctx={ctx4.subCtx(o => o.orderDate)} unitText={ago(o.orderDate)} readOnly={true} />

<ValueLine ctx={ctx4.subCtx(o => o.requiredDate)} unitText={ago(o.requiredDate)} onChange={() => this.forceUpdate()} />

<ValueLine ctx={ctx4.subCtx(o => o.shippedDate)} unitText={ago(o.shippedDate)} hideIfNull={true} readOnly={true} />

<ValueLine ctx={ctx4.subCtx(o => o.cancelationDate)} unitText={ago(o.cancelationDate)} hideIfNull={true} readOnly={true} />

function ago(date: string | null | undefined) {

    if (!date)

        return undefined;

    return moment(date).fromNow(true);

}

*Note how requiredDate, the only editable ValueLine, requires a forceUpdate in onChange to refresh the unitText string.*

Another small improvement is to change the color of the state label.

In other UI frameworks is often cumbersome to change the style of encapsulated components, but with react and the rest operator it is quite simple to expose properties that give you total control to customize the appearance of parts of a component.

In this case we can change the color property, of the style property, of the generated label for the read-only ValueLine showing an enum value, just using valueHtmlAttributes prop.

<ValueLine ctx={ctx4.subCtx(o => o.state)} readOnly={true} valueHtmlAttributes={{ style: { color: stateColor(o.state) } }} />

function stateColor(s: OrderState | undefined) {

    if (!s)

        return undefined;

    switch (s) {

        case "New":

        case "Ordered": return "#33cc33";

        case "Shipped": return "#0066ff";

        case "Canceled": return "#ff0000";

    }

}

### Using EntityTable

Finally, the main important part of the Order entity are their lines, each containing the product, quantity, unit price, optional discount and the calculated sub-total.

Resembling a printed Invoice, the UI control that makes more sense is a table. Luckily Signum.React contains a new EntityTable component that is perfect for adding, removing and modifying entities in an MList of an Entity using a tabular form.

Let’s see how we use it:

<EntityTable ctx={ctx2.subCtx(o => o.details)} onChange={() => this.forceUpdate()}

    columns={EntityTable.typedColumns<OrderDetailEmbedded>([

    {

        property: a => a.product,

        headerHtmlAttributes: { style: { width: "40%" } },

        template: dc => <EntityLine ctx={dc.subCtx(a => a.product)} onChange={() => this.handleProductChange(dc.value)} />

    },

    {

        property: a => a.quantity,

        headerHtmlAttributes: { style: { width: "15%" } },

        template: dc => <ValueLine ctx={dc.subCtx(a => a.quantity)} onChange={() => this.forceUpdate()} />

    },

    {

        property: a => a.unitPrice,

        headerHtmlAttributes: { style: { width: "15%" } },

        template: dc => <ValueLine ctx={dc.subCtx(a => a.unitPrice)} readOnly={true} />

    },

    {

        property: a => a.discount,

        headerHtmlAttributes: { style: { width: "15%" } },

        template: dc => <ValueLine ctx={dc.subCtx(a => a.discount)} onChange={() => this.forceUpdate()} />

    },

{

        header: "SubTotalPrice",

        headerHtmlAttributes: { style: { width: "15%" } },

        template: dc => <FormGroup ctx={dc}>

            <FormControlStatic ctx={dc}>

                {numbro(subTotalPrice(dc.value)).format()} €

            </FormControlStatic>

        </FormGroup>

    },

])} />

EntityTable stats with the ctx, like any other property component, and contains an optional property columns. If the property is not set, the component will automatically show all the public properties in the order they are declared.

The columns property is typically initialized using the helper method EntityTable.typedColumns that gives a statically-typed context to the declaration inside, improving auto-completion.

Column contains an optional ‘property’ field. It’s a lambda to a field. When set, the component is able to automatically determine the Header (property NiceName) and the template (most appropriate component for the property type). You can however customize this properties.

In our case, we are showing the product, quantity, unitPrice and discount but customizing the header style (to set recommended widths), and the templates, to recalculate subTotalPrice.

The last column, SubTotalPrice, is completely synthetic, not backed by any property, so we are forced to set the header and template manually. There we use the subTotalPrice function, defined like:

function subTotalPrice(od: OrderDetailEmbedded) {

    return (od.quantity || 0) \* (od.unitPrice || 0) \* (1 - (od.discount || 0));

}

Also we use some new components, FormGroup is used for placing a label and a column side by side, following the StyleContext preferences. FormControlStatic properly centers the value.

Finally, when we set the product, we automatically set the unitPrice and quantity using handleProductChange.

handleProductChange = (detail: OrderDetailEmbedded) => {

    detail.quantity = 1;

    detail.unitPrice = 0;

    this.forceUpdate();

    if (detail.product)

        Navigator.API.fetchAndForget(detail.product)

            .then(p => detail.unitPrice = p.unitPrice)

            .then(() => this.forceUpdate())

            .done();

}

### Localization

Signum framework assumes that some common concepts of your application, like Entity type names and properties, will be localized and takes advantage of this convention in SearchControl, property components etc…

When creating custom string literals in the UI, however, we will need to create and use our own localizable string. In this case let’s add a new value to OrderMessage enum in C#:

public enum OrderMessage

    {

        …

        SubTotalPrice,

    }

And use it in the column header of the UI:

{

        header: OrderMessage.SubTotalPrice.niceToString(),

        headerHtmlAttributes: { style: { width: "15%" } },

        …

    }

That’s all from the developer point of view! At some point a translator will realize, using Translation extension, that there are some new strings to translate, and suggest plausible translations using Azure translate or alternative translation providers.

## Overriding Operations

Operations are Signum way of defining actions that can be done on entities. This operations are defined in the server side, can be authorized, will be logged and provide out of the box a UI.

There are five types of operations:

* **Construct:** Creates a new entity, when defined, is used from the [+] button in a SearchControl or property component (like EntityLine). If not, the default client-side constructor is used.
* **ConstructFrom:** Creates a new entity from another entity. In the UI appears as a button on an entity frame, or a contextual menu on a Search Control row.
* **ConstructFromMany:** Creates a new entity from many other ones. In the UI appears only as a contextual menu on a Search Control row.
* **Execute:** Modifies an Entity. Appears as a button on an entity frame, or a contextual menu on a Search Control, by default only if the operation is Lite=true (sending only a reference to the entity, not a full entity with potential changes).
* **Delete:** Physicallyremoves an entity from the database. Available as a red button on an entity frame, or a contextual menu on the Search Control.

[AutoInit]

public static class OrderOperation

{

    public static ConstructSymbol<OrderEntity>.Simple Create;  
    public static ConstructSymbol<OrderEntity>.From<CustomerEntity> CreateOrderFromCustomer;

    public static ConstructSymbol<OrderEntity>.FromMany<ProductEntity> CreateOrderFromProducts;

    public static ExecuteSymbol<OrderEntity> SaveNew;

    public static ExecuteSymbol<OrderEntity> Save;

    public static ExecuteSymbol<OrderEntity> Ship;

    public static ExecuteSymbol<OrderEntity> Cancel;

    public static DeleteSymbol<OrderEntity> Delete;

}

All this default UI behavior can be customized using OperationSettings. Let’s see how:

### Customizing Construct operation

Typically construct operation doesn’t need any arguments, in our case however an Order can accept an optional CustomerEntity. Here is the server side:

new Construct(OrderOperation.Create)

    {

        ToStates = { OrderState.New },

        Construct = (args) =>

        {

            var customer = args.TryGetArgC<Lite<CustomerEntity>>()?.Retrieve();

            return new OrderEntity

            {

                Customer = customer,

                ShipAddress = customer?.Address.Clone(),

                State = OrderState.New,

                Employee = EmployeeEntity.Current,

                RequiredDate = DateTime.Now.AddDays(3),

            };

        }

    }.Register();

On the client-side, we can override the Construct operation in SouthwindClient.tsx like this:

Operations.addSettings(new ConstructorOperationSettings(OrderOperation.Create, {

    onConstruct: coc =>

    {

        return Finder.find({ queryName: CustomerQuery.Customer }).then(c => {

            if (!c)

                return undefined;

            return coc.defaultConstruct(c);

        });

    }

}));

This code uses Finder.find to open a SearchModal to look for a Customer.

Since Customers can be Person or Employees we use CustomerQuery.Customer, a custom query that makes a union of the two tables. Alternatively we could have chosen the type first, and then open a SearchModal specific for the selected type. Check DynamicQueryCore.Manual and CustomerQuery.Custom to know more about queries that union tables.

Finder.find returns a Promise<Lite<T> | undefined>, returning the selected lite if a row is selected, or undefined if the modal is closed. In our code we return undefined if the search is cancel, but we could create the company anyway.

Finally we can use coc (for ConstructOperationContext) defaultConstruct to invoke the normal behavior (calling the server) but this time passing the selected customer as a parameter.

#### Operations with additional arguments:

Typically the server deserializes data from the client using the type information provided in the arguments of Action in the Controller.

When using defaultConstruct, or any other method that calls generic Actions defined in OperationController (Signum Framework), this information is not available.

For convenience, OperationController and GetArg/TryGetArg helper methods are able to infer the types of some arguments: strings, numbers, Enums, Lite<Entity>, Entity, ModelEntity and arrays. For any other type, like custom DTOs, you’ll need a custom controller.

### Customizing ConstructFromMany operation

The operation OrderOperation.CreateOrderFromProducts defines how to construct an order from many products. In the UI, once in Product table, you can select some of them and using the contextual menu create a new Order.

Similarly to the simple constructor, this also accepts an optional Customer. And similarly we can override the UI to search for it:

Operations.addSettings(new ContextualOperationSettings(OrderOperation.CreateOrderFromProducts, {

    onClick: coc => {

        return Finder.find({ queryName: CustomerQuery.Customer })

            .then(c => {

                if (!c)

                    return;

                return coc.defaultContextualClick(c);

            }).done();

    }

}));

Note: Look that in this case we use a ContextualOperationSettings instead of a ConstructOperationSettings.

### Customizing Execute operation

As a third example, the order has a Ship operation that requires an optional ShipDate, if not provided, then Now is assumed. Here we override the behavior when the operation is used from the entity itself (onClick) or from a contextual menu (contextual.onClick):

const selectShippedDate = () => ValueLineModal.show({

    type: { name: "datetime" },

    initialValue: moment().format(),

    labelText: OrderEntity.nicePropertyName(a => a.shippedDate)

});

Operations.addSettings(new EntityOperationSettings(OrderOperation.Ship, {

    onClick: (eoc) => {

        selectShippedDate()

            .then(date => eoc.defaultClick(date))

            .done();

    },

    contextual: {

        onClick: coc => {

            selectShippedDate()

                .then(date => coc.defaultContextualClick(date))

                .done();

        }

    }

}));//Ship

In both cases we use a ValueLineModal to ask for the ShipDate. This component is a ready-to-use Modal for asking for simple values and supports all the options of ValueLine.

Note: Note how in this case we use EntityOperationSettings. Also, there is no need to override OrderOperation.CreateOrderFromCustomer, or OrderOperation.Delete for Order because the default behavior is all right, but in case of need it would be very similar to this example, since both are allowed as entity frame button and contextual menu.

### Calling Operations from Client Side

Sometimes, instead of customizing the behavior of operation buttons that where place there automatically, you want to hide the default button.

Operations.addSettings(new EntityOperationSettings(OrderOperation.Cancel, {

    isVisible: eoc => false

}));

And add your own custom closer to some related functionality, like extra buttons in a SearchControl.

extraButtons={sc => [

<button className="btn btn-info" onClick={this.handleShipClick}>

{Reflection.symbolNiceName(OrderOperation.Ship)}

</button>

]}

In this case, you can still use all the server-side APIs already provided by OperationController using Operations.tsx API namespace.

var ctx = this.props.ctx;

   Operations.API.executeEntity(ctx.value, OrderOperation.Cancel)

       .then(pack => ctx.frame!.onReload(pack))

       .done();

Very often just calling the Server-side and reloading is not enough, having to include validation and a success notification message. In order to get the full package you can use the following code:

var eoc = EntityOperationContext.fromTypeContext(ctx, OrderOperation.Cancel);

  EntityOperations.defaultExecuteEntity(eoc);

## SearchControl

The SearchControl is the most powerful component in Signum.React. It lets the user query the database, starting from a simple starting with just a query defined in logic, and then navigate to the main entities of this query or other related entities. The user is also able to create new entities or, depending on the installed extensions, save the query as a user query, export to excel, generate charts, create email and word templates, etc.

The component consist in a fully functional grid with many capabilities:

* **Order columns**: By clicking in the column header we can toggle ascending/descending. Pressing Shift at the same time allows defining secondary order criteria.
* **Add filters:** By pressing [Add filter] we can create a new filter manually specifying the query token operation, and value.   
  Trick: we can be faster by right-clicking in a column header (automatic query token), or even on a table cell (automatic query token and value).
* **Manage columns:** The columns on the SearchControl can be added and removed. The changes not only hide and remove the columns visually, the underlying SQL query also gets simpler/ more complex.
  + **Remove:** By right-clicking on the column header.
  + **Rename:** By right-clicking on the column header.
  + **Add:** By right-clicking between column headers we will be able to specify the query token for the new column.
  + **Reorder:** By drag-and-dropping column header.

### QueryToken

The configuration of the SearchControl is built on top of the concept of QueryToken. A QueryToken is a simple linear expression that lets the user navigate through entity tables following foreign keys.

This expressions are simple enough for showing them to the end user using just a sequence of ComboBoxes as the UI, but powerful enough to express most of the queries. They are intensively used in Filters and Columns of the SearchControl, as well as Charting, Emails and Word Templates, etc…

Assuming a query is defined in logic like:

sb.Include<OrderEntity>()

    .WithQuery(dqm, () => o => new

    {

        Entity = o,

        o.Id,

        o.State,

        o.Customer,

        o.Employee,

        o.OrderDate,

        o.RequiredDate,

        o.ShipAddress,

        o.ShipVia,

    });

Here are some examples of QueryToken:

* **“State”:** The column “State” of the query.
* **“Entity.State”:** The property “State” of the order, accessible through “Entity” column in the query.
* **“Employee.Name”:** The name of the Employee, note how a Join will be needed in the database.
* **“ShipAddress.Country”:** Access to the country of the ShipAddress, no join will be needed because is an embedded entity.
* **“OrderDate.Year”:** Accessing the year of the OrderDate, will be translated to the equivalent T-SQL code.
* **“Entity.Details.Count”:** Accessing the OrderDetails table with the lines of the order, and counting how many there are.
* **“Entity.Details.Element.Product”:** Accessingeach particular line of the Order, potentially multiplying the number of rows in the result, fact that will be shown with a yellow warning message.
* **“Entity.Details.Element.UnitPrice”:** Accessingthe same particular line of the Order, the join will be unified with the previous token.
* **“Entity.Details.Element2.Product”:** Accessing each particular line of the Order, but this time doing a new, independent, join with the table. .
* **“Entity.Details.Any.Product”:** Allows checking conditions of the lines without multiplying the number of rows. Only available for filters. Other possibilities are All/NotAny/NotAll.

For the sake of UI simplicity, query tokens deviate slightly from C# / LINQ expressions:

* **Nullability:** All the QueryTokens are nullable by default and nullability propagates naturally.
* **Lite<T>:** No QueryToken ever retrieves a full entity, only Lite<Entity>. Accessing the Entity property is also automatic.

### Embedded SearchControl

Contrary to how we developed entity components, using composition with a battery of independent property components like ValueLine, EntityLine, etc… SearchControl contains a bunch of interrelated functionality that it’s better controlled with configuration options and a set of extension points.

Inside of an entity component, or any other react component, it’s often a good idea to include a small SearchControl optionally filtering by some parent entity. The SearchControl component has one important property, findOptions, that represents the configuration of the search control including:

* **queryName**: The name of the query as registered in logic. Typically associated with an Entity type of a Enum value to keep everything statically typed.
* **filters**: List of filters that will be shown by default at the beginning. Each containing a queryToken, filterOperation and value.
* **columns/columnsMode**: List of columns that will be added/removed/replaced from the original query, depending on columns mode.
* **orders**: List of initial orders for the query, each containing query token and a OrderType (ascending or descending).

When a SearchControl is embedded in another entity, typically you want to do two things: filter by a particular column, and hide the column from the results, since now is a constant value.

<SearchControl findOptions={{

    queryName: OrderEntity,

    filterOptions: [{ columnName: "Customer", operation: "EqualTo", value: ctx.value }],

    columnOptionsMode: "Remove",

    columnOptions: [{ columnName: "Customer" }]

}} />

This pattern is so common, that there are two additional properties in the findOptions to make it slightly simpler: parentColumn and parentValue.

<SearchControl findOptions={{

    queryNamen: OrderEntity,

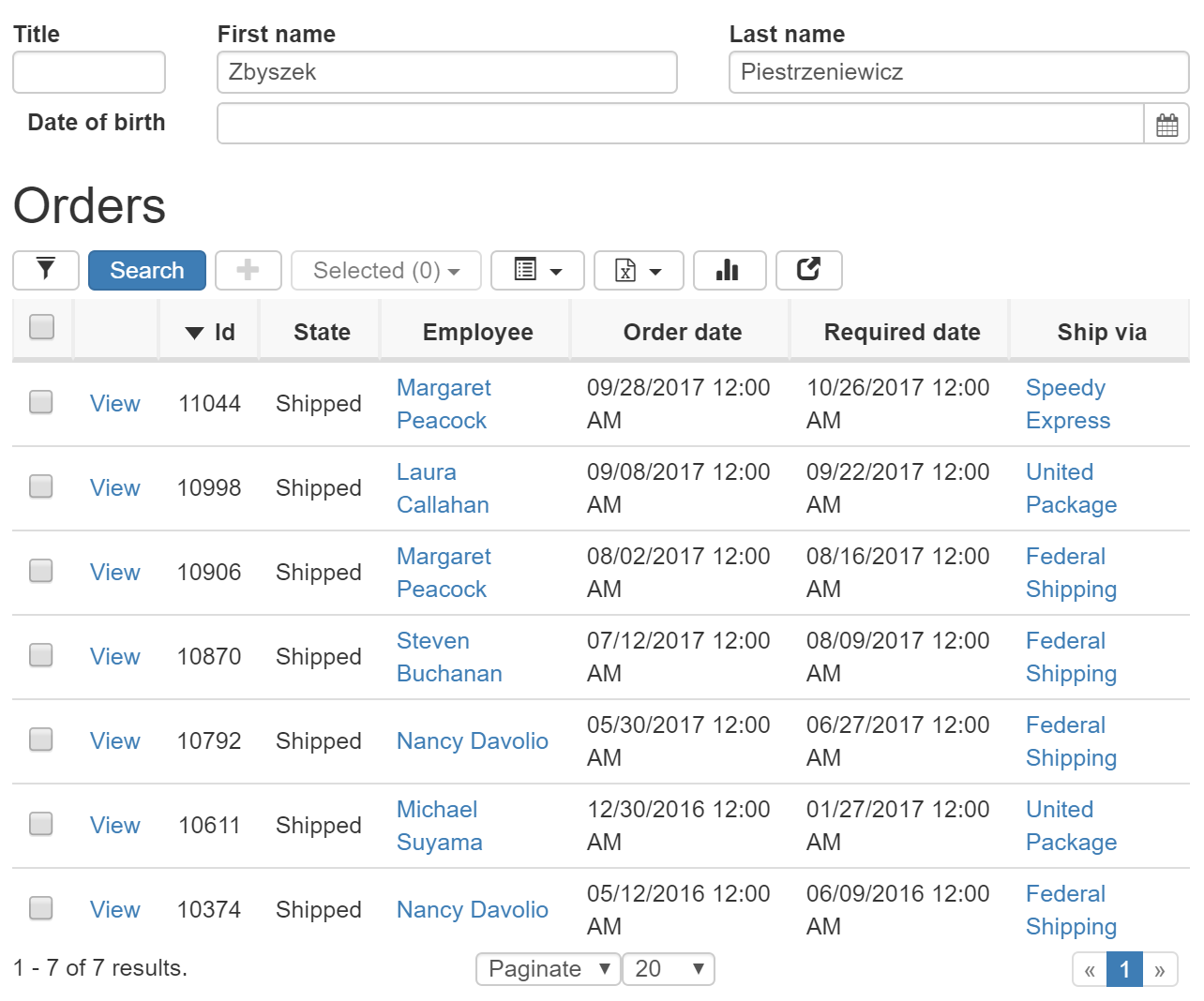
    parentColumn: "Customer",

    parentValue: ctx.value

}} />

Also note that a SearchControl does not contain any title, but we can add it taking advantage of the translations (singular and plural) of any entity type.

<h2>{OrderEntity.nicePluralName()}</h2>

This is the end result inside of the Person component (Person.tsx): 

The SearchControl itself contains many configuration options to customize its behavior and appearance, for example:

* Customize appearance in the grid:
  + formatters
  + entityFormatters
  + rowAttributes
  + allowSelection
  + navigate
* Control button bar:
  + extraButtons
  + showBarExtension
  + showBarExtensionOptions
  + showFooter
  + create
  + largeToolbarButtons
* Control filters/orders/columns:
  + showFilters
  + showFilterButton
  + showSimpleFilterBuilder
  + allowChangeColumns
  + allowChangeOrder
* Other SearchControl behavior:
  + searchOnLoad
  + avoidAutoRefresh
  + throwIfNotFindable

The SearchControl also provides some callbacks that will be called on some actions, like:

* onSearch
* onResult
* onCreate
* onDoubleClick
* onSelectionChanged
* onFiltersChanged
* onNavigated

### QuerySettings

All this configuration options can only be specified for a custom SearchControl that is under your control, but often the SearchControl is inside a SearchPage or SearchPopup that is opened by the framework. Using QuerySettings we can customize every SearchControl that shows one particular query.

#### CellFormatters

In our example we would like to add a CellFormatter for one particular PropertyRoute: The picture of a CategoryEntity.

This means that any time any QueryToken ends up in this property, whether it’s added from the user or was already defined in the default query, our custom code will be used to render the value.

const maxDimensions: React.CSSProperties = { maxWidth: "96px", maxHeight: "96px" };

Finder.registerPropertyFormatter(CategoryEntity.propertyRoute(ca => ca.picture),

       new Finder.CellFormatter((cell: FileEmbedded) =>

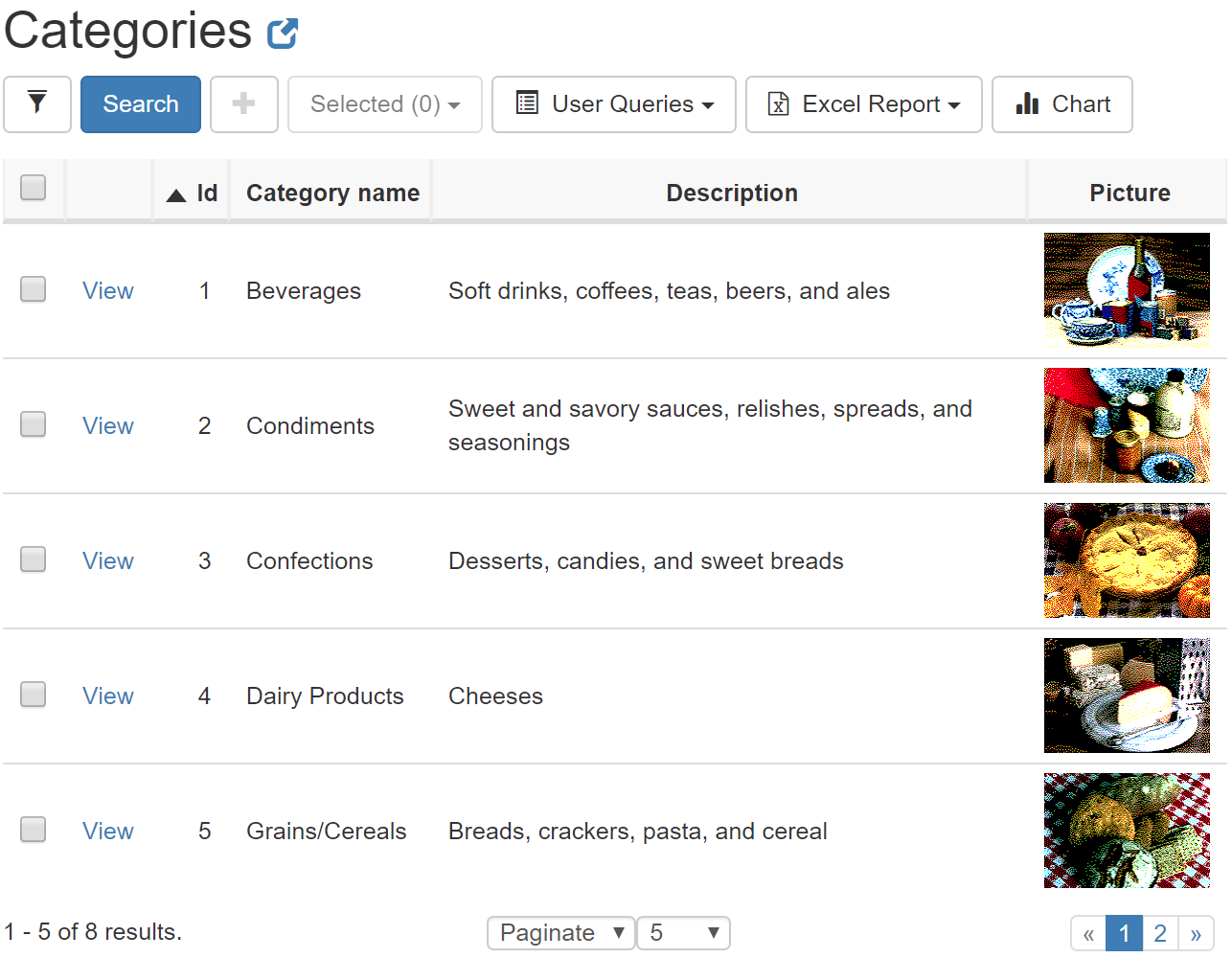
<img style={maxDimensions} src={"data:image/jpeg;base64," + cell.binaryFile} />

));

Let’s analyze this code:

* Finder.registerPropertyFormatter requires a PropertyRoute and a CellFormatter.
* We can create a PropertyRoute with propertyRoute method in any Type<T>, and a statically-typed lambda.
* Finally, a CellFormatter only requires a lambda from the data in the column to the HTML that we want to generate, in this case an image with fixed dimensions and a data-url.

This is the final result:



#### RowAttributes and HiddenColumns

QuerySettings also allows us to change the htmlAttributes of the TR element of each table row.

Finder.addSettings({

    queryName: OrderEntity,

    rowAttributes: (row, columns) => {

        var state = row.columns[columns.indexOf("State")] as OrderState;

        var color = state == "Canceled" ? "darkred" :

            state == "Shipped" ? "gray" :

                "black";

        return { style: { color: color } };

    }

});

In this case we’re changing the color (text color) of the row depending on the state of the order.

Unfortunately, the user is free to remove the state column from the UI, making our code buggy. We can prevent this by adding hiddenColumns property as well.

Finder.addSettings({

    queryName: OrderEntity,

    hiddenColumns: [{ columnName: "State" }],

    rowAttributes: (row, columns) => {

        var state = row.columns[columns.indexOf("State")] as OrderState;

        var color = state == "Canceled" ? "darkred" :

            state == "Shipped" ? "gray" :

                "black";

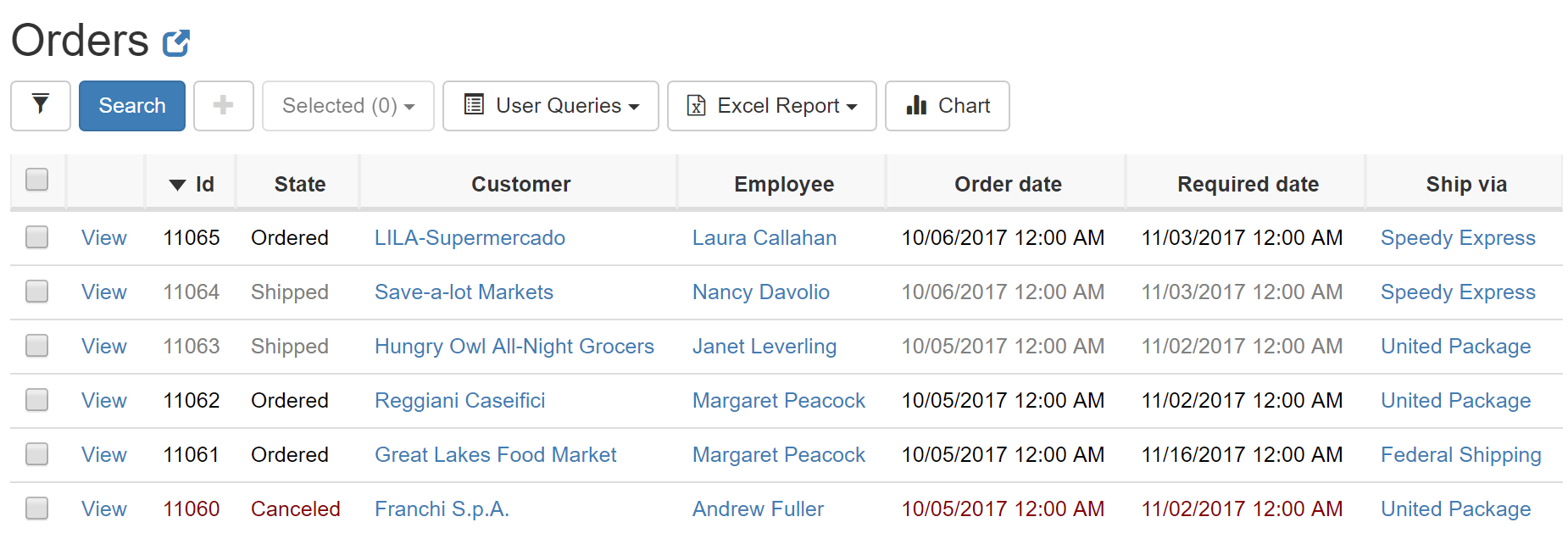
        return { style: { color: color } };

    }

});

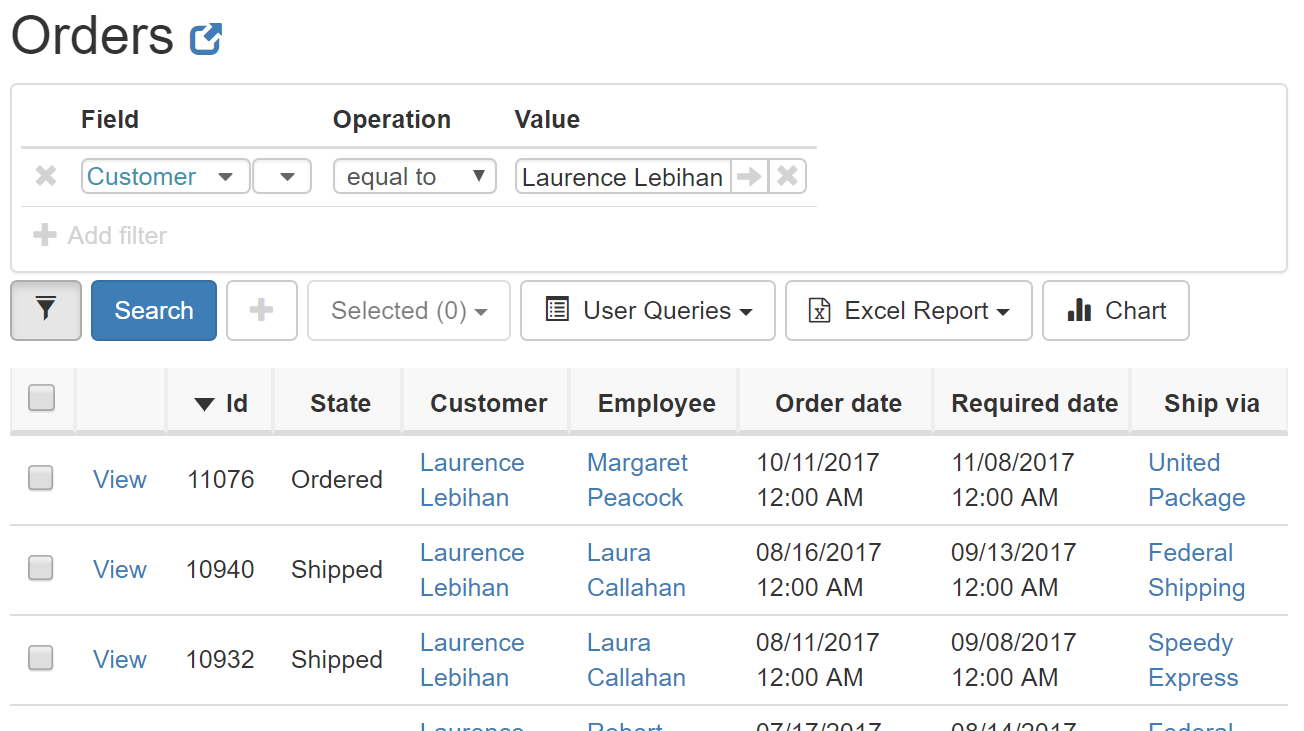
This way the State column will be requested to the database even if the user removes it.

Let’s see the result:

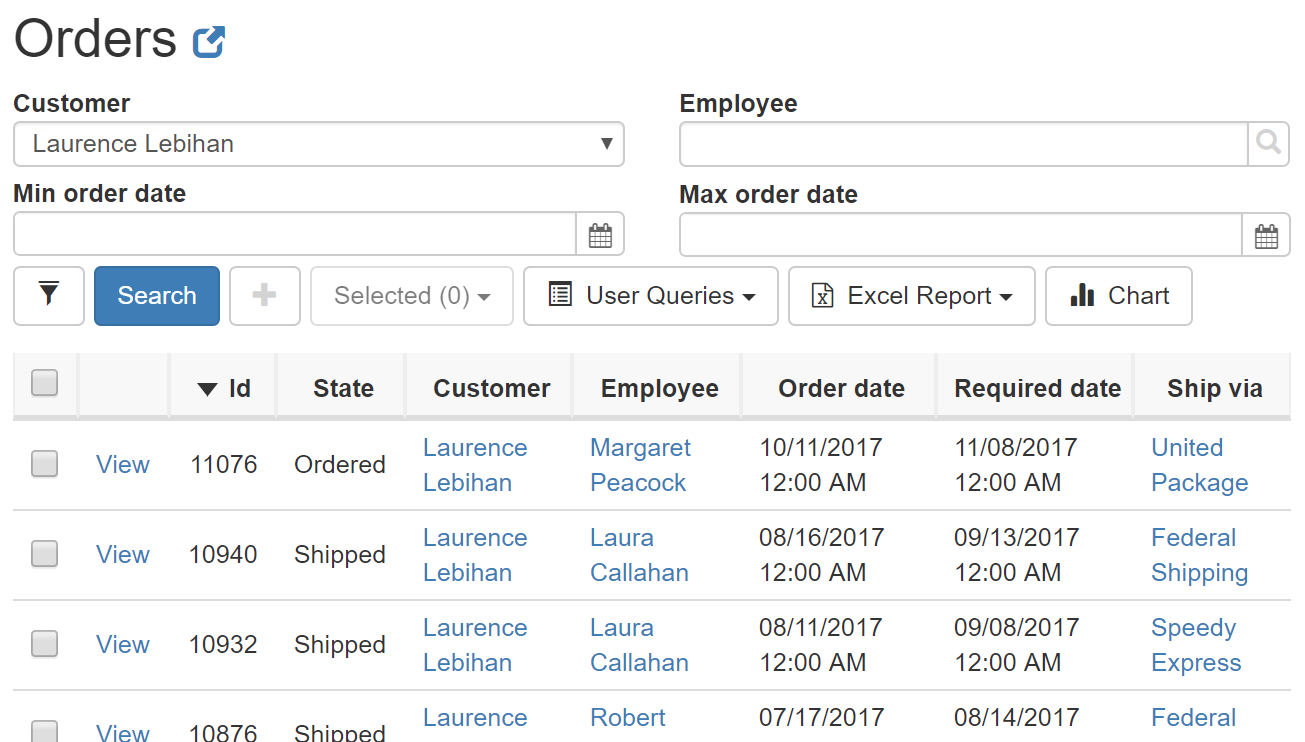


#### SimpleFilterBuilder

The SearchControl provides out of the box a FilterBuilder that, using QueryTokens, operations and values can express a wide range of conditions.



For SearchControls that are going to be used intensively (like in a Call Center) or by one-time users (like a Catalog) could make sense to make a more traditional UI.



The property to control this in QuerySettings is called simpleFilterBuilder. Let’s see how to use it:

First we start with a ModelEntity that represents the field that we will like to have in the filters:

[Serializable]

public class OrderFilterModel : ModelEntity

{

    [ImplementedBy(typeof(PersonEntity), typeof(CompanyEntity))]

    public Lite<CustomerEntity> Customer { get; set; }

    public Lite<EmployeeEntity> Employee { get; set; }

    [DaysPrecissionValidator]

    public DateTime? MinOrderDate { get; set; }

    [DaysPrecissionValidator]

    public DateTime? MaxOrderDate { get; set; }

}

And a component to layout the different components:

export default class OrderFilter extends React.Component<{ ctx: TypeContext<OrderFilterModel> }>   
 implements ISimpleFilterBuilder {

    render() {

        const ctx = this.props.ctx.subCtx({ formGroupStyle: "Basic" });

        return (

            <div className="form-vertical">

                <div className="row">

                    <div className="col-sm-6">

                        <EntityCombo ctx={ctx.subCtx(o => o.customer) } />

                        <ValueLine ctx={ctx.subCtx(o => o.minOrderDate) } />

                    </div>

                    <div className="col-sm-6">

                        <EntityLine ctx={ctx.subCtx(o => o.employee) } />

                        <ValueLine ctx={ctx.subCtx(o => o.maxOrderDate) } />

                    </div>

                </div>

            </div>

        );

    }

}

This component looks identical to any entity component, but also implements ISimpleFilterBuilder. This requires us to implement the getFilters method:

getFilters(): FilterOption[] {

    const result: FilterOption[] = [];

    const val = this.props.ctx.value;

    if (val.customer)

        result.push({ columnName: "Customer", value: val.customer });

    if (val.employee)

        result.push({ columnName: "Employee", value: val.employee });

    if (val.minOrderDate)

        result.push({ columnName: "OrderDate", value: val.minOrderDate, operation: "GreaterThanOrEqual" });

    if (val.maxOrderDate)

        result.push({ columnName: "OrderDate", value: val.maxOrderDate, operation: "LessThan" });

    return result;

}

What this method does is translating the our OrderFilterModel into a list of FilterOptions, in order to keep the server side similar, and allow the search control to have a button to toggle to the more general and powerful FilterBuilder.

Finally we need to register our SimpleFilterBuilder:

Finder.addSettings({

    queryName: OrderEntity,

    simpleFilterBuilder: (qd, fop) => {

        return <OrderFilter ctx={TypeContext.root(OrderFilterModel.New())} />;

    }

});

This approach works O.K. when the SearchControl is open clean, without filters.

In case of an embedded SearchControl with default filters, or a SearchPage opened from a QuickLink, drill-down on a Chart or any other case where default filters are expected, these filters will be lost.

A simple solution will be to disable the SimpleFilterBuilder whenever there are filters:

Finder.addSettings({

    queryName: OrderEntity,

    simpleFilterBuilder: (qd, fop) => {

        if (fop.length > 0)

            return undefined;

        return <OrderFilter ctx={TypeContext.root(OrderFilterModel.New())} />;

    }

});

A slightly more accurate approach will be to try to fit the provided filters into the model and, if all the filters are able to be expresses in the more limited but user-friendly SimpleFilterBuilder, then use it. For this we need to define an extract function:

static extract(fos: FilterOptionParsed[]) {

    const filters = fos.clone();

    const result = OrderFilterModel.New({

        customer: extractFilterValue(filters, "Customer", "EqualTo"),

        employee: extractFilterValue(filters, "Employee", "EqualTo"),

        minOrderDate: extractFilterValue(filters, "OrderDate", "GreaterThanOrEqual"),

        maxOrderDate: extractFilterValue(filters, "OrderDate", "LessThan"),

    });

    if (filters.length)

        return undefined;

    return result;

}

Every call to extractFilterValue fiends a filter with the provided query token and operation, if found returns the value and removes the filter from the list.

At the end of the OrderFilterModel initialization two things could happen:

* If the filters are now empty, all the filters where able to be expresses with the OrderFilterModel, so we can return it.
* Otherwise we return undefined, meaning better to use the general FilterBuilder.

We will need then to adapt the code in SouthwindClient accordingly:

Finder.addSettings({

    queryName: OrderEntity,

    simpleFilterBuilder: (qd, fop) => {

        const model = OrderFilter.extract(fop);

        if (!model)

            return undefined;

        return <OrderFilter ctx={TypeContext.root(model) }/>;

    }

});

## Appendix 1: C# to TypeScript CheatSheet:

Here is an small list of typical code snippets translated from C# to TypeScript

| **C#** | **TypeScript** |
| --- | --- |
| enum OrderState { Ordered, Cancelled } | type OrderState = "Ordered" | “Cancelled” |
| class OrderEntity : Entity | interface OrderEntity exends Entity |
| OrderEntity o; | var o : OrderEntity; |
| o.OrderNumber | o.orderNumber |
| o.ToString() | getToString(o) |
| o.GetType() == typeof(OrderEntity) | o.Type == OrderEntity.TypeName |
| o.State = OrderState.Ordered | o.state = "Ordered" |
| o.TotalAmount.ToString("0.0") | numbro(o.totalAmount).format("0.0") |
| o.CreationDate = DateTime.Now | o.creationDate = moment().format() |
| OrderState.Ordered.NiceToString() | OrderState.niceName("Ordered") |
| typeof(OrderEntity).NiceName() | OrderEntity.niceName() |
| OrderMessage.Please.NiceToString() | OrderMessage.Please.niceToString() |
|  |  |