Performance

There are a few ways to make sure that a NGClient runs fast, some are provided as configuration options for webcomponents in there spec files others are best practices

Webcomponent properties

For performance reasons (and security) the spec file of a webcomponent should tell us what property should (and can) be sent to the server.

This page: Specification (.spec file) under Data synchronization we list the values the pushToServer attribute of a model property can have. By default the property is not allowed to be pushed to the server and therefore also not watched (by an angular $watch). Besides that you can also specify "allow" this also means that we don't add a watch on it, but the component pushes the value itself when it knows that it is changed. Dataproviders with the auto-apply directive work like that (Provided directives, filters, services and model values). Then the auto-apply directive will push data when the directive is triggered (dom onchange event). Components can also use the servoyApi.apply("dataprovider_propertyname") when the need to program it out. Using the above will not result in a loss of performance.

If you do want other properties to be sent to the server because you want to access them in scripting you can use one of the 2 values "shallow" or "deep". Shallow is quite cheap that just a reference check, but when you have a complex object that has properties (or an array) which doesn't change itself but it changes the content (sets a property of that object) then you need to use "deep". This has a performance penalty because angular now needs to create a copy of the original object and compare all the time the full structure of that object. This is not a problem when there are only a few on one page but if if you have many of them, for example in a portal like component, then this can count up.

Above we talk about properties going from client to server, NGClient also provides a way to optimize the server to client communication of properties. Component developers can use this if you have a very complex component with many properties that must all be watched somehow (by a angular directive or directly as a $watch in code). In the link (or controller) function of a component directive where you get the servoy model pushed of the properties, you can add a special function on the model object:

The model change listener

```
Object.defineProperty($scope.model,$sabloConstants.modelChangeNotifier, {configurable:true,value:function
(property,value) {
                                switch(property) {
                                        case "borderType":
                                                $svyProperties.setBorder(element,value);
                                                break;
                                        case "background":
                                        case "transparent":
                                                $svyProperties.setCssProperty(element,"backgroundColor",$scope.
model.transparent?"transparent":$scope.model.background);
                                                break;
```

You need to include the $sabloConstants in your component which will have a property "modelChangeNotifier" that property is the name of the function that is then added with a Object.defineProperty call to the $scope.model object.

That function will receive the property name and the value when something is pushed from the server. Then you can do stuff for that specific property like setting some css or adding a class.

Besides adding it you also need to remove it when the $scope is destroyed and also make sure that the first values are pushed:

Destroy/InitialPush

```
          var destroyListenerUnreg = $scope.$on("$destroy", function() {
                                destroyListenerUnreg();
                                delete $scope.model[$sabloConstants.modelChangeNotifier];
                        });
                        // data can already be here, if so call the modelChange function so that it is
initialized correctly.
                        var modelChangFunction = $scope.model[$sabloConstants.modelChangeNotifier];
                        for (key in $scope.model) {
                                modelChangFunction(key,$scope.model[key]);
                        }
```

A full example: https://github.com/Servoy/servoy-client/blob/af95c443a8258c54ee494e87e1126c8ed66e7831/servoy_ngclient/war/servoydefault/label/label.js

Tableview/Portals

If your solutions uses tableviews in readonly mode only then NGClient has a special property that can be set in the onsolution open method: APP_UI_PROPERTY.TABLEVIEW_NG_OPTIMIZED_READONLY_MODE.

enabling the readonly mode.

```
application.putClientProperty(APP_UI_PROPERTY.TABLEVIEW_NG_OPTIMIZED_READONLY_MODE,true)
```

If you enable that then all existing tableviews will be in readonly mode, except the forms/portals which have the **ngReadOnlyMode** set to "false".

With the APP_UI_PROPERTY.TABLEVIEW_NG_OPTIMIZED_READONLY_MODE all the textfield and typeahead fields from tableviews will be replaced with a very light webcomponent. Buttons and Labels work the same so a click on them will fire the action. Currently only texfield and typeahead are replaced, others like combobox and datefield should still be done.

Of course this means that the tableview can't really be used to directly edit the data, also using readonly textfields and enabling this property is faster then using all labels (because labels are not optimized because on action and so on need to work)

Web resource optimizations

On the admin page under the ngclient settings there is one option that can improve initial load performance when deployed:

servoy.ngclient.enableWebResourceOptimizer: This enabled the grouping of various js and css files, so the the browser only sees a few js and css files which are a combination of all the standard and component javascript and css files.

Things to avoid

- Try to minimize the usage of controller.recreateUI(), Using the solutionModel is not a problem if you use it upfront to make all your forms the way you want. But try to avoid regenerating the forms all the time, this was quite cheap for the SmartClient, was already quite a bit heavier for a WebClient, but for the NGClient it has even more performance implications because quite a bit of logic needs o constantly be pushed to the client (besides a new template)
- Try to avoid a lot of nesting of forms, for example use a portal component instead of tabpanel that has a tableview form again. Or using a form for just a toolbar that could be quite an easy component.
- Try to avoid touching elements or forms when you are not (planning to) show them. Setting properties is not a problem, but calling api on an element of a form that is not shown means that we have to quickly create it (in a hidden div) so that the element is really alive.