LUND UNIVERSITY

The PML Editor: User's Manual

Johnsson, Björn A; Weibull, Gunnar; Magnusson, Boris

2017

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Johnsson, B. A., Weibull, G., & Magnusson, B. (2017). *The PML Editor: User's Manual*. (Technical Report; No. 102). Department of Computer Science, Lund University.

Total number of authors:
3

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

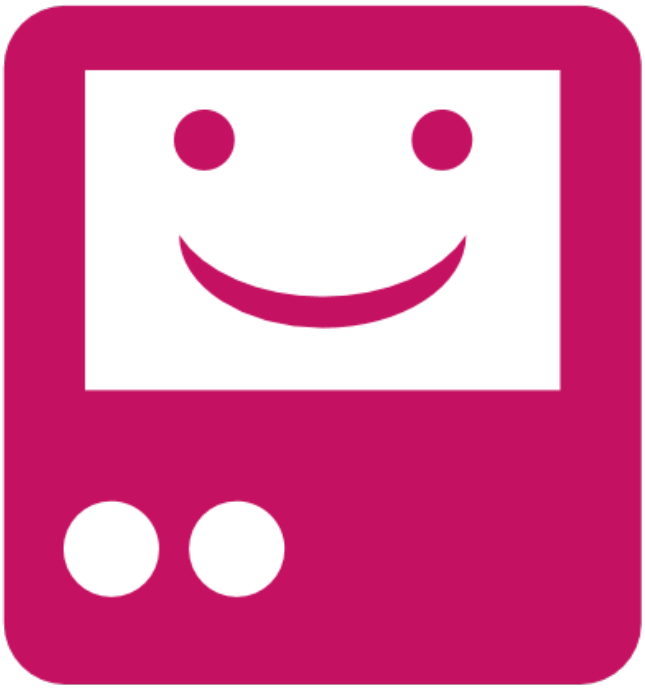
LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

The PML Editor

User's Manual

25 May 2016



Gunnar Weibull
Boris Magnusson
Björn A. Johnsson

Technical Report, LU-CS-TR:2017-253
ISSN 1404-1200, Report 102, 2017
Lund University

Report 102, 2017
LU-CS-TR:2017-253

ISSN 1404-1200

Department of Computer Science
Lund University
Box 118
SE-211 00 Lund
Sweden

Typeset using Pages version 6.1
Illustrated using Google Drawings

© 2017 The authors

Table of Contents

Table of Contents 3

What can I do with the PML Editor? 5

Introduction 5

Example applications 5

What is a PML file? 6

Quick start guide 7

Overview 23

PML Components 23

Overview of the program 26

The control panel 27

The network palette 29

List of network units 30

The canvas 31

List of components in the canvas 33

The facts and notifications panel 35

List of notifications 35

List of facts 35

Creating components 36

Creating components with links 36

Creating static components 39

Creating windows 40

Creating notifications and facts 42

Links 43

What is a link? 43

Link types 44

Creating links between existing components 46

Getting an overview of links 48

Editing and removing existing links 51

List of links 56

Using facts 59

Setting default values 59

Enabling/disabling and hiding/showing components 59

Linking components that can't be connected directly 60

Layout 61

Moving components 61

Layout types 62

Areas and layout nesting 65

Spacing with empty labels 67
Setting the size of components 68
Borders on Android 69
Creating a custom Android toolbar 70
Running PML files 71
Introduction 71
Installing AndroidPUIDI 72
Connectivity 73
Usage Overview 73
Running PML files from the editor 74
Installing descriptions from a PalCom Browser 74
Custom parts 75
Creating custom parts 75
Loading custom parts into the PML Editor 76
Installing custom parts on AndroidPUIDI 76
Troubleshooting 77
Network devices don’t appear 77
I can’t link components that I want to be able to link 77

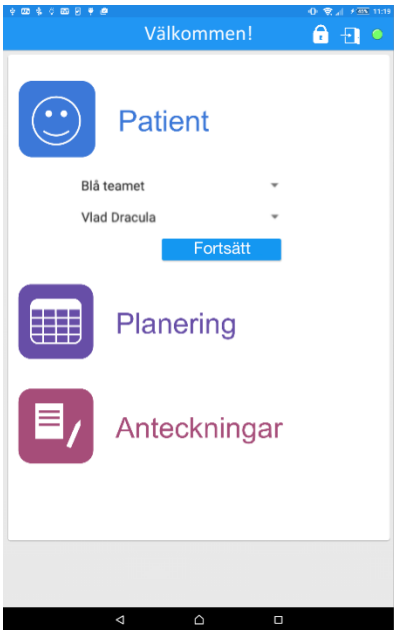
What can I do with the PML Editor?

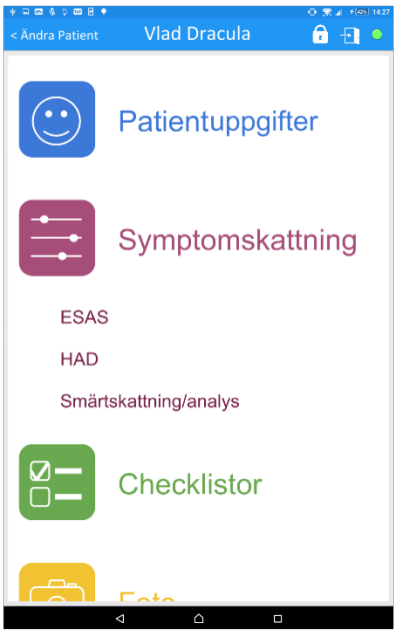
Introduction

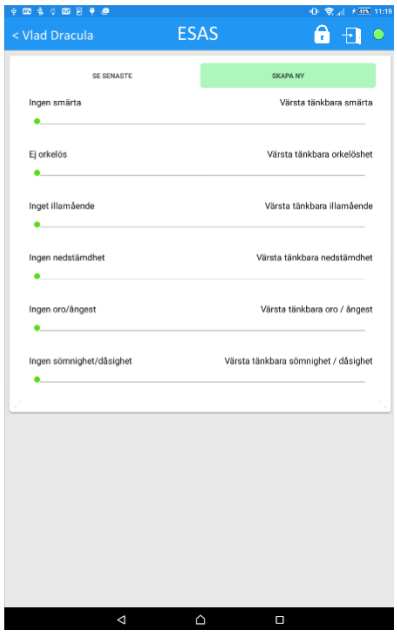
In short, PML is a convenient way of creating applications which communicate with devices on a PalCom network. With the PML Editor, you can design the way your application will look, as well as what the application should do.

Example applications

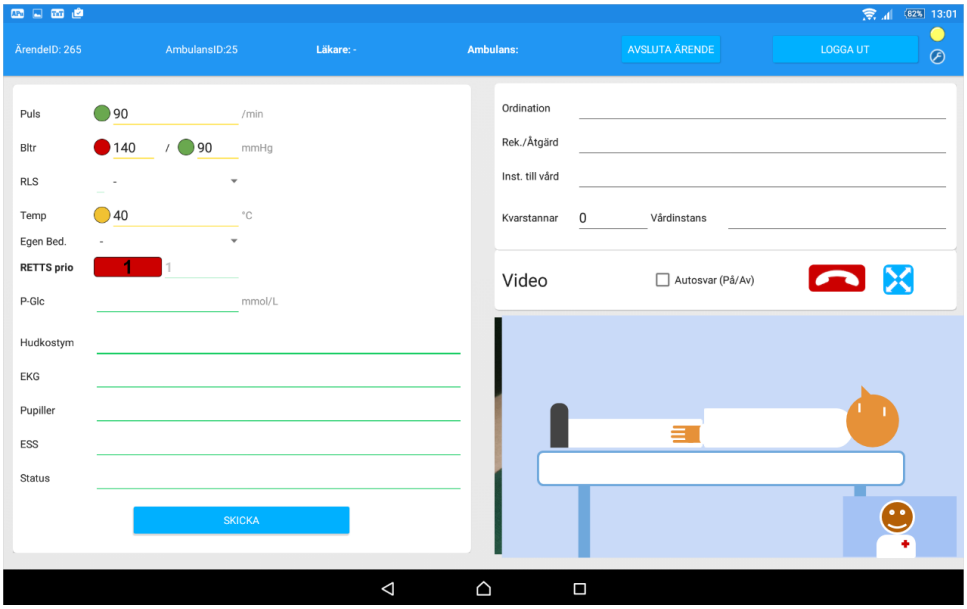
PML has been used to create an application for advanced healthcare in the home, which nurses can use for tasks such as planning, getting and filling in data about patients etc:







Another example of an application that is currently being developed in PML is an app which enables ambulance personnel to communicate with doctors via video, as well as send over data about the patient:



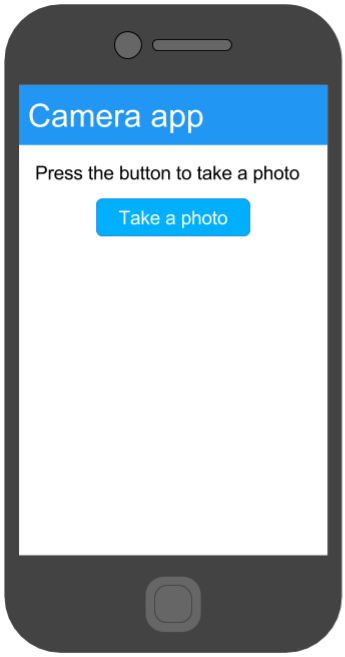
What is a PML file?

In the PML Editor, you create *PML files*. In practical terms, a PML file contains information about what an application should *look like* and how it should *function.* In other words, the PML file lists the components of the application (e.g. buttons, text fields…), where they are placed in the application and how they are linked (e.g. “the button should send a command”).

The PML file thus contains all the information about the application, but to *actually run it*, you need to use a *PML back-end*. The currently available back-end is an application for Android, which you can install PML files on. Installing and running PML files is as simple as clicking a button in the PML Editor.

Quick start guide

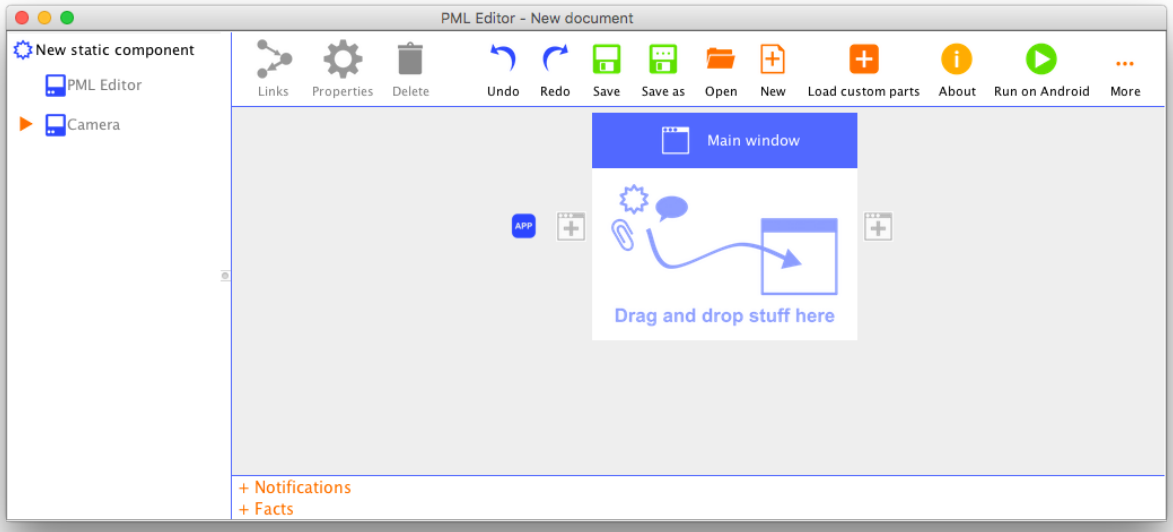
This part of the manual will guide you through the creation of a simple camera app, highlighting the basics of the Editor as we go. The app will communicate with a (fake) camera, and will include a button to take a photo. When a photo is taken, the app will switch to a new window, and show the taken photo.



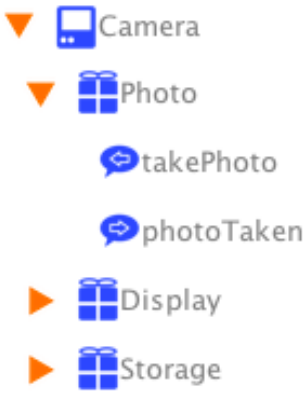
First window

Second window

When starting the editor, it will look something like this:

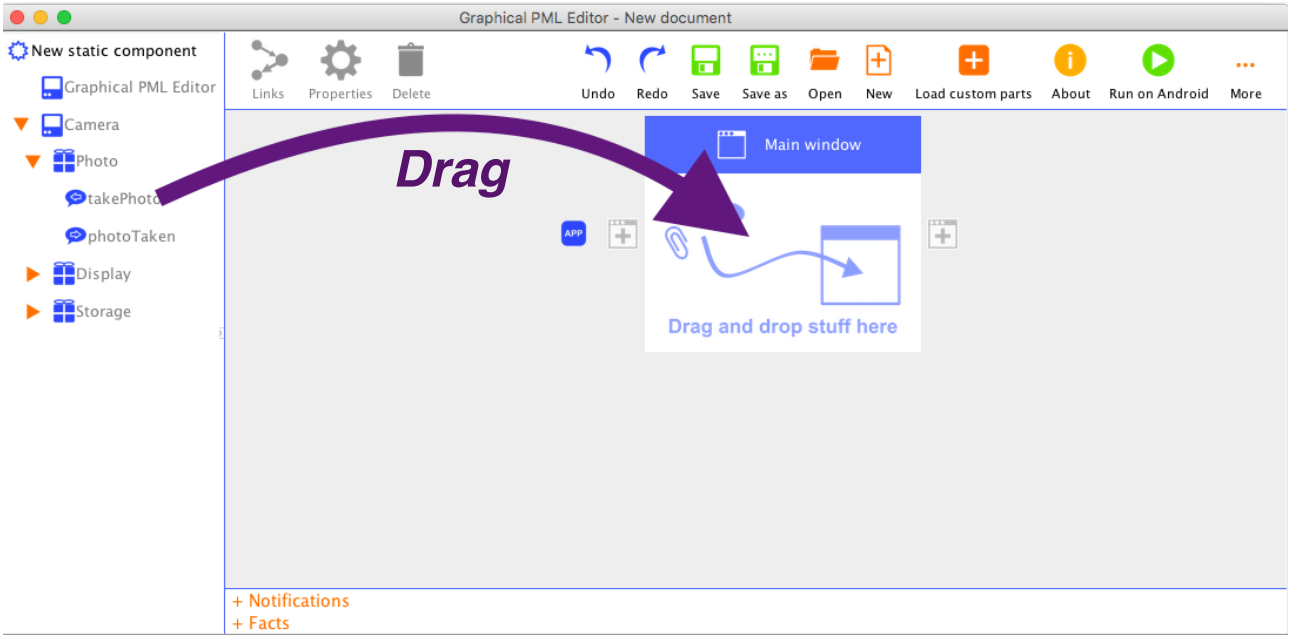


To the left, we see a list of network devices. The device we want to use is the Camera device. We click the orange arrow to see what services the Camera offers.

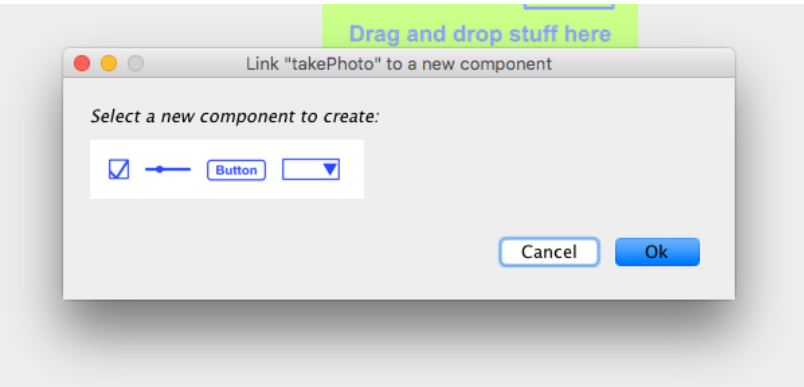


The services are shown as small packages, and represent a functionality of the device. The *photo*-service contains commands to take a photo (*takePhoto*) and to be notified when a photo is taken (*photoTaken*), shown as speech bubbles. The arrow of the command indicates whether the command is *to* the device (left-facing arrow) or *from* the device (right-facing arrow).

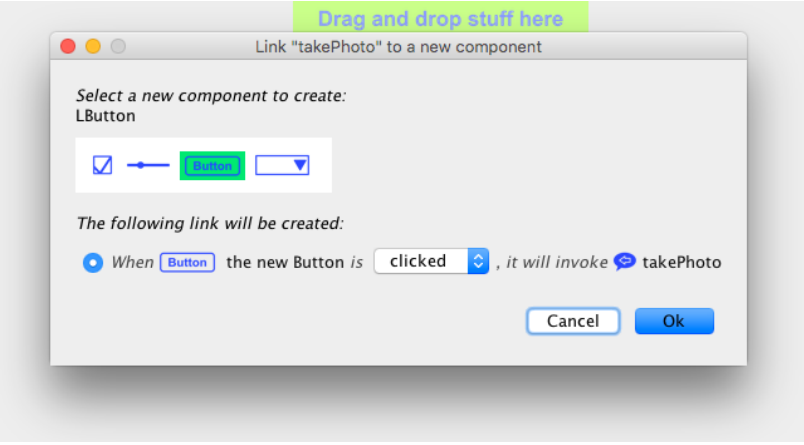
In order to create a button to send the *takePhoto* command (to tell the camera to take a photo), we simply drag the takePhoto-command to the canvas.



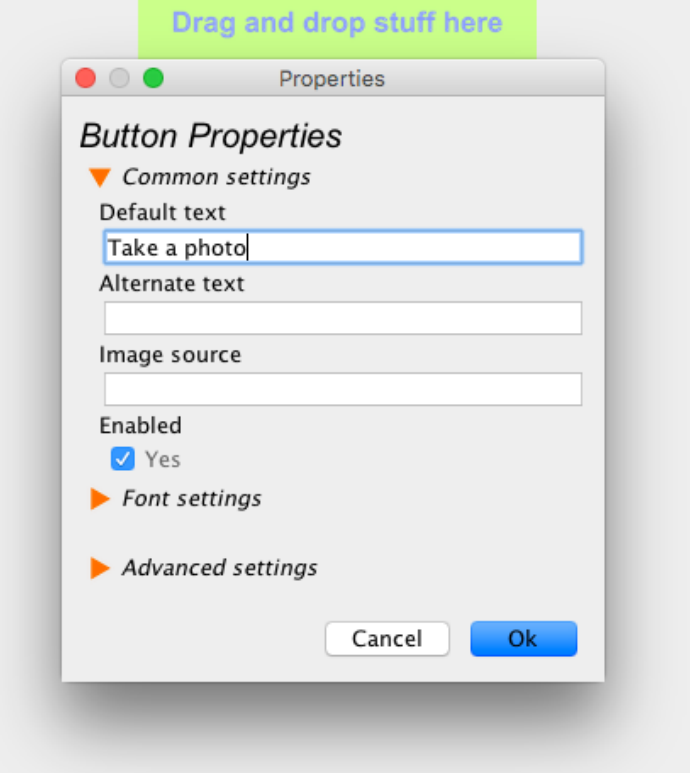
Dragging the *takePhoto*-command and dropping it in the canvas brings up this dialog:



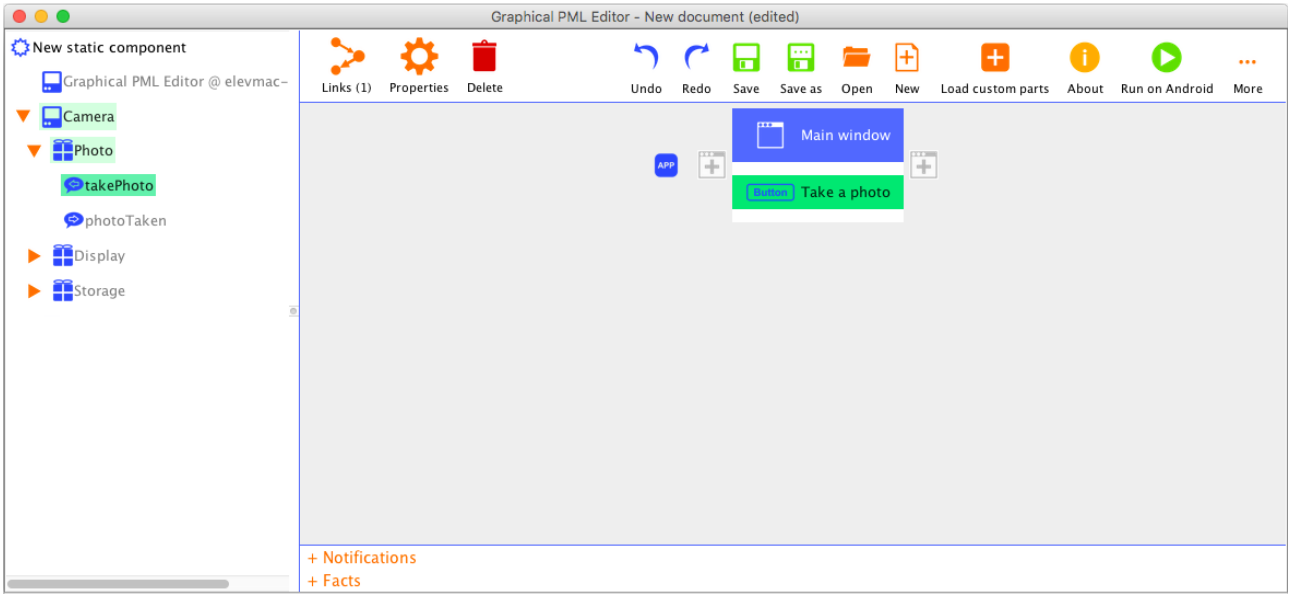
The dialog presents some options for what we can use for sending the command. We click the button icon:

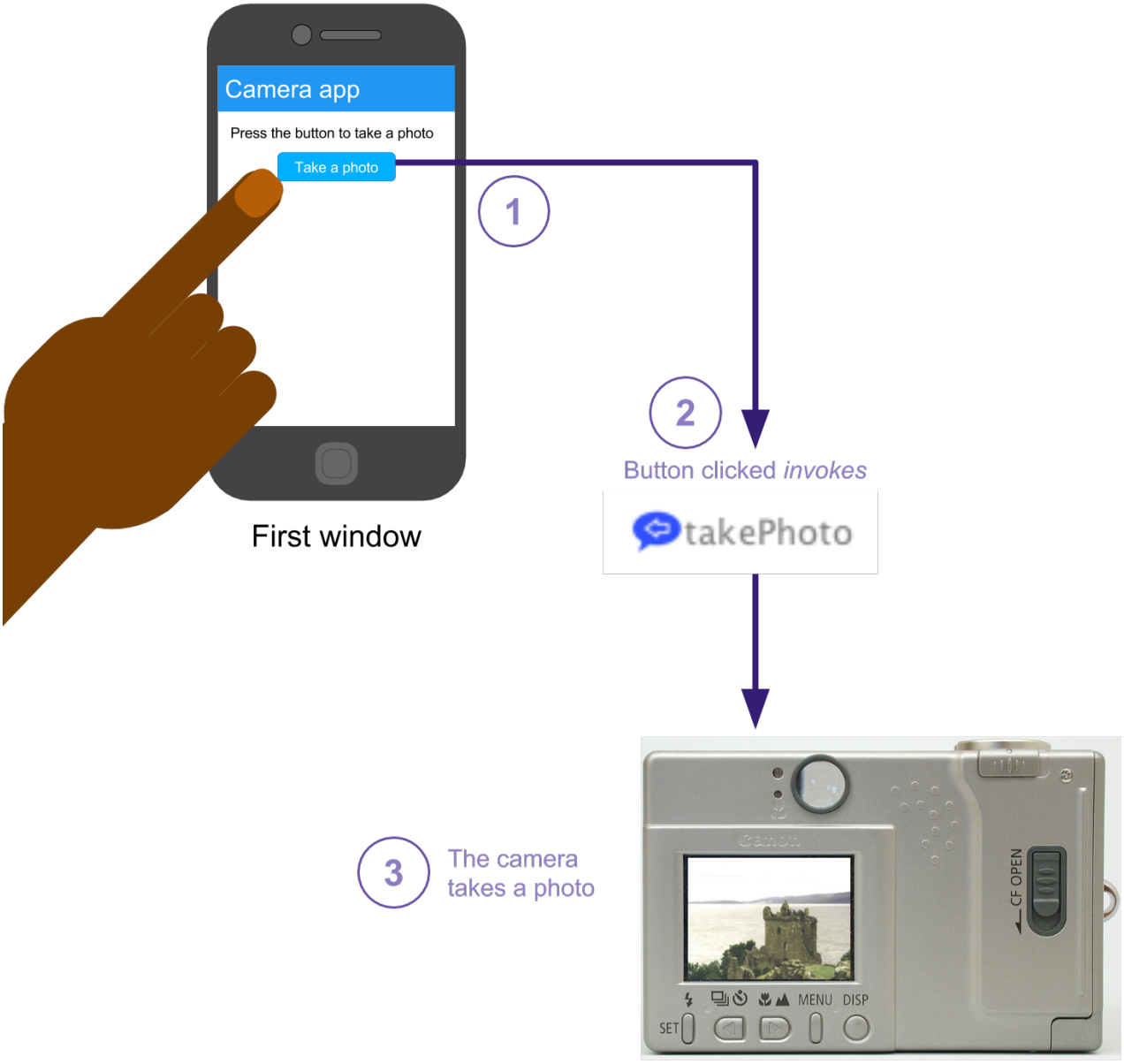


After clicking the button icon, we get an explanation of the link that will be created: When the new button is clicked, the command will be sent. Clicking OK brings up the last dialog, which lets us set properties for the new button. We set the default text to “Take a photo” and click OK.



Now, we have a button to send the *takePhoto*-command. We can confirm that the link has been created by clicking the button. The button becomes bright green, indicating it’s selected. We see that the takePhoto-command becomes green as well. This indicates that it’s linked to the button. The parent components of *takePhoto* become light green, to make the linked command easier to locate.

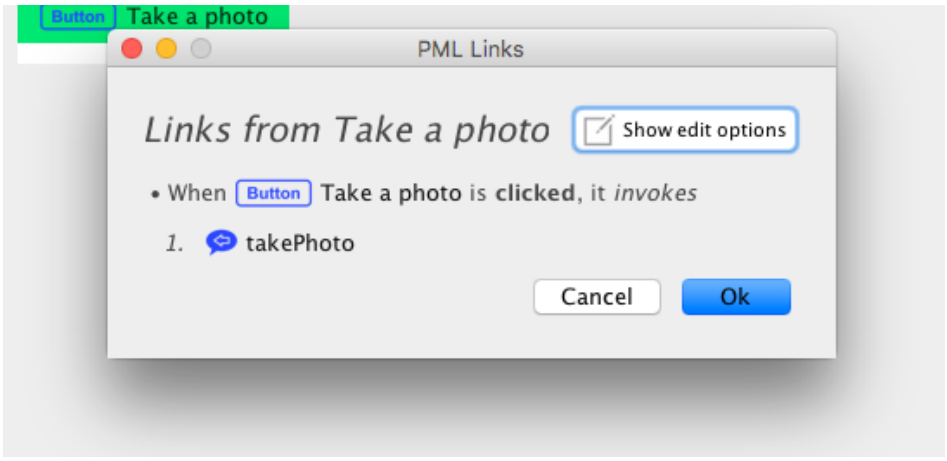




Overview of the application so far.

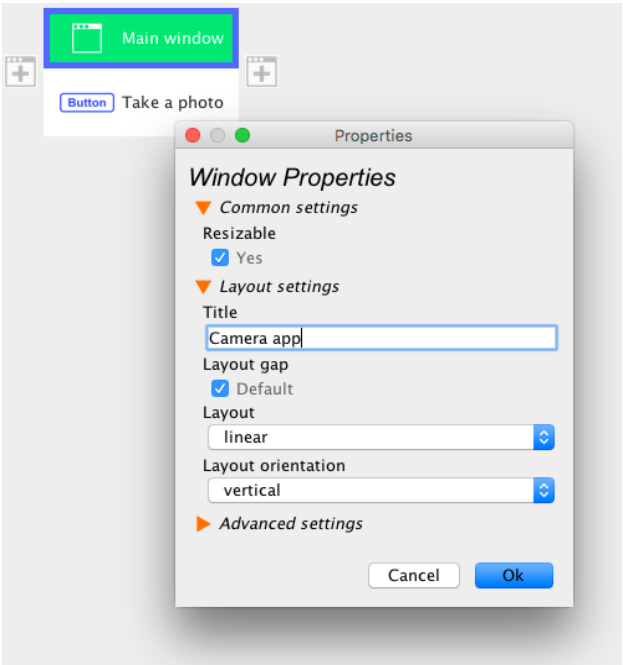
The three buttons in the left part of the top bar concern the selected component (in this case the button). The “Links”-button gives a more detailed description of the links to and from the button, the “Properties”-button shows the dialog where we can change properties for the button (the same one we changed the button text in earlier). The “Delete”-button deletes the button and all its links.

Clicking the “Links”-button brings up the following dialog:

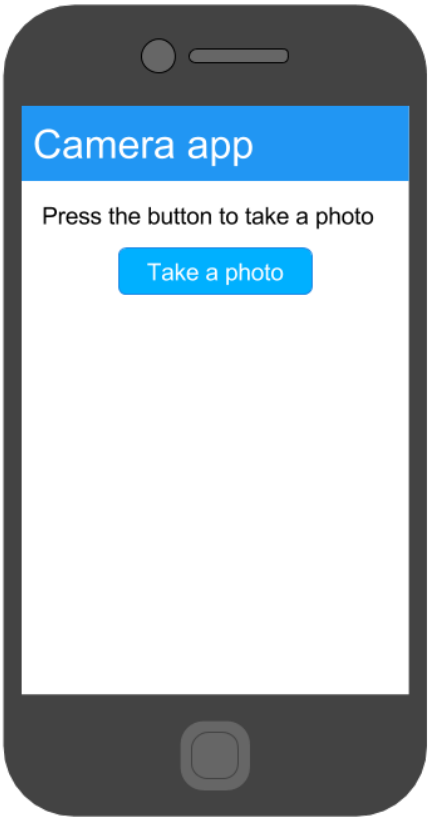


Clicking the takePhoto-command in this dialog makes it flash in the GUI, which is very handy for locating linked components in larger applications. Clicking the “Show edit options”-button brings up more options for the links, such as deleting them.

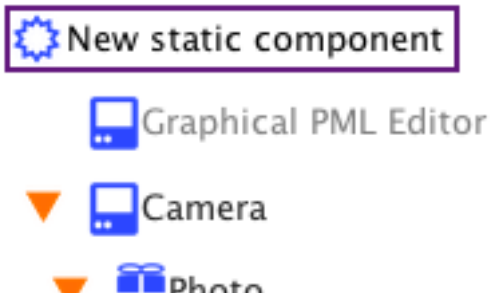
To change the window title, we can select the window in the canvas and click the “Properties”-button, and change the “Title”-property. Click OK to save the changes.



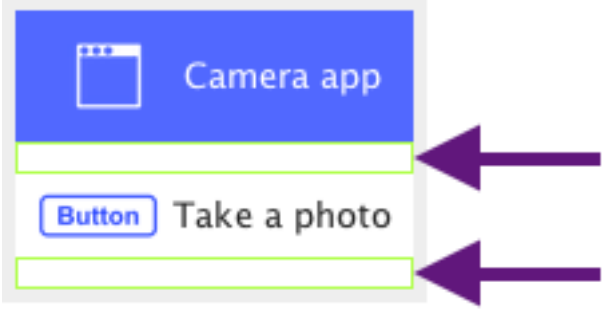
The final thing we want to add to the first window of the application is the text “Press the button to take a photo”.



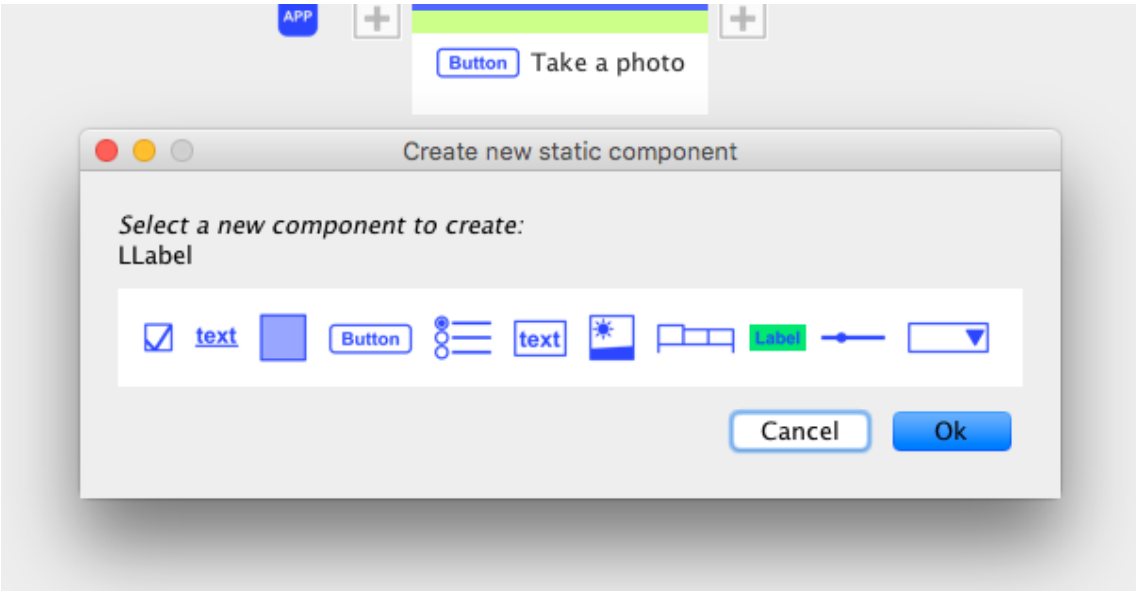
This text label is considered a *static* component, since it won’t be linked to anything (the text won’t be changed). Adding static components is done similarly to how we added the button, but this time we will drag the “New static component”-label:

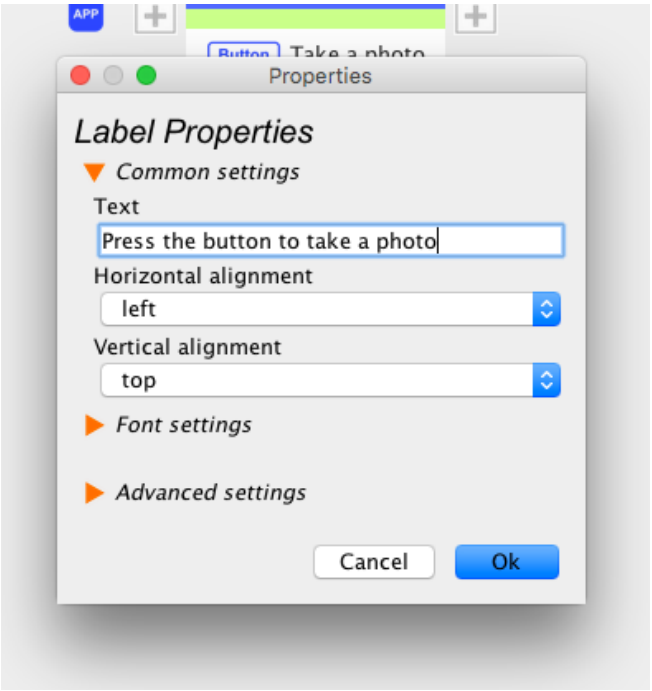


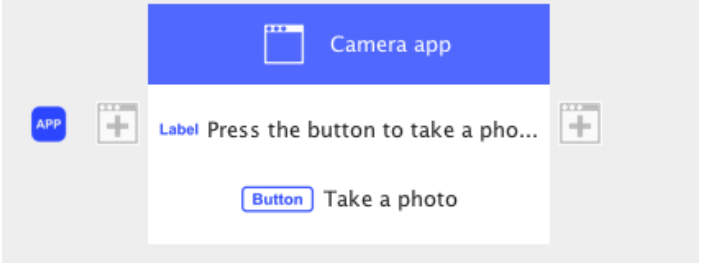
When dragging the “New static component”-label, we notice that two spots in the canvas light up. These are places where we can create a new component. We drag and drop the “New static component”-label to the top one, since we want the text to be above the button.



Dropping the “New static component”-label brings up similar dialogs to when we created the button. We select *Label* in the first dialog, and set the text to “Press the button to take a photo” in the second:

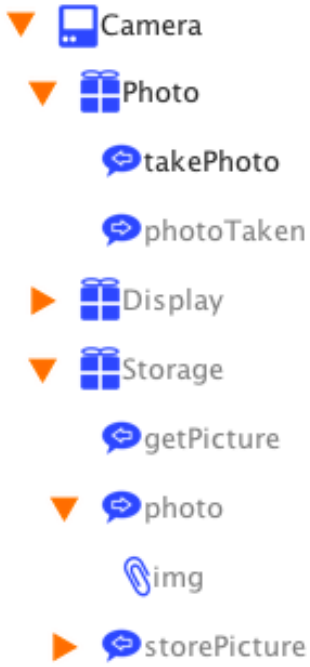






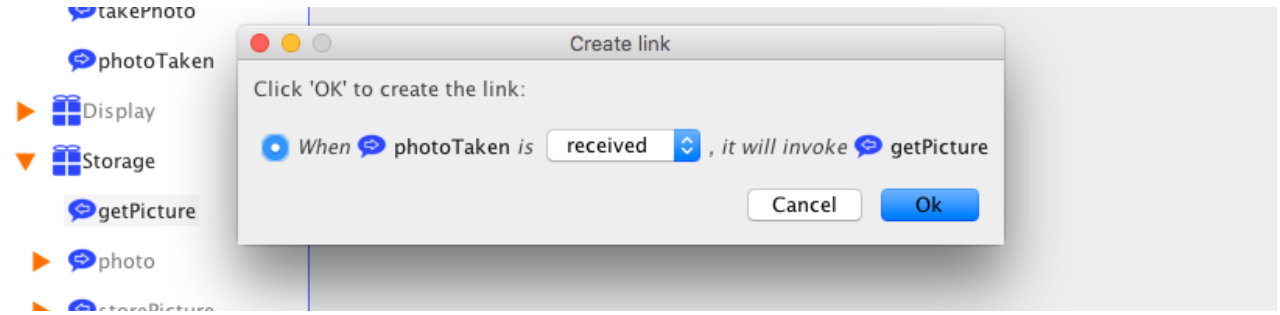
Now that we are finished with the creation of the first window of the application, it's time to start with the second:

To figure out how to get the picture from the camera, we take a closer look at the *Storage*-service of the camera.



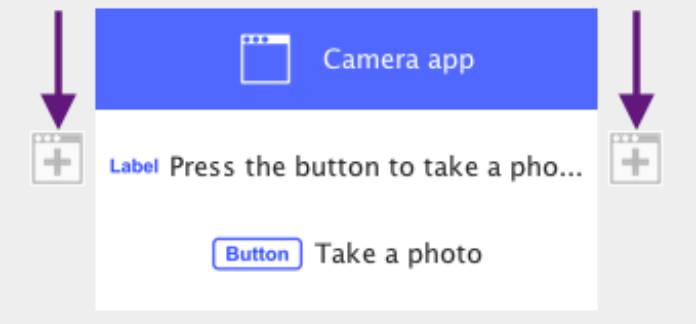
The *Storage*-service has a command called *getPicture* which can be sent to the camera (left-facing arrow), and a command called *photo* with a *parameter* called *img* which can be sent to the application. Commands can have parameters in order to send something, like an image or a text (the commands in themselves can only be used to tell the device to do something or for the device to tell us something has happened).

We want the camera to send the image to us when it has been taken, and thus we need to create a link between the *photoTaken*-command (which lets us know that the image we told the camera to take has been taken) and the *getPicture*-command (which tells the camera to send the picture to us). This is done by dragging and dropping *photoTaken* onto *getPicture* (or *getPicture* onto *photoTaken*). We get the following dialog, which lets us know what the link will do:

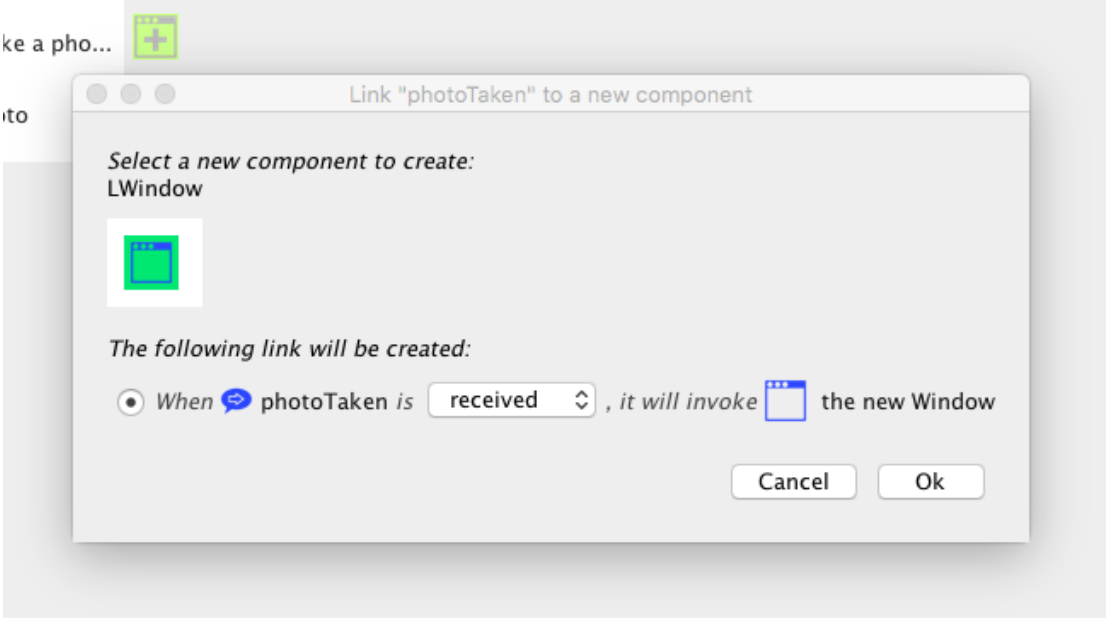


This is what we want to happen, so we click OK to create the link.

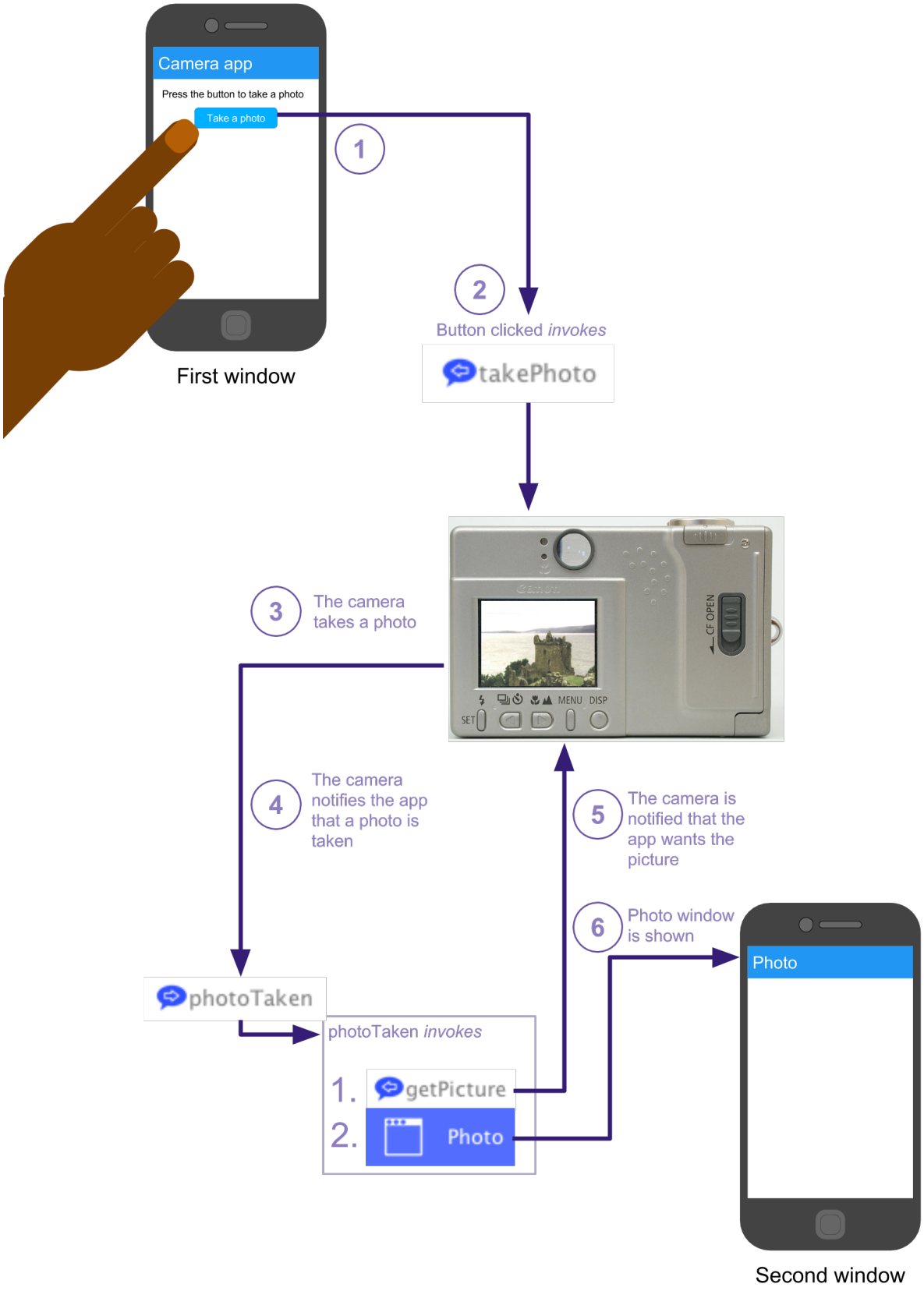
Next, we want to display the image in a new window. Firstly, we need to create the window. Since we want the application to switch to the new window when we have taken a photo, we drag the *photoTaken*-command to one of the small “Add window”-icons next to the existing window (it doesn’t matter which one).



This brings up the usual dialog. When the *photoTaken*-command is received, the new window will be shown.

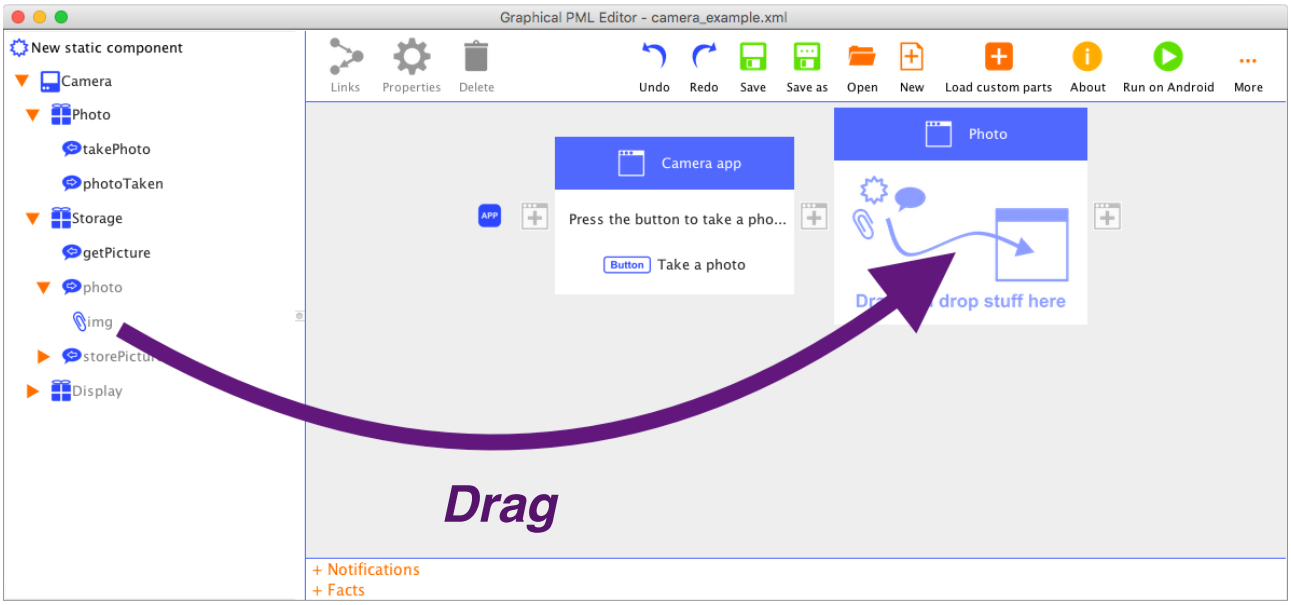


We click OK to create the window with the link, and set “Photo” as the title in the window properties dialog that appears.

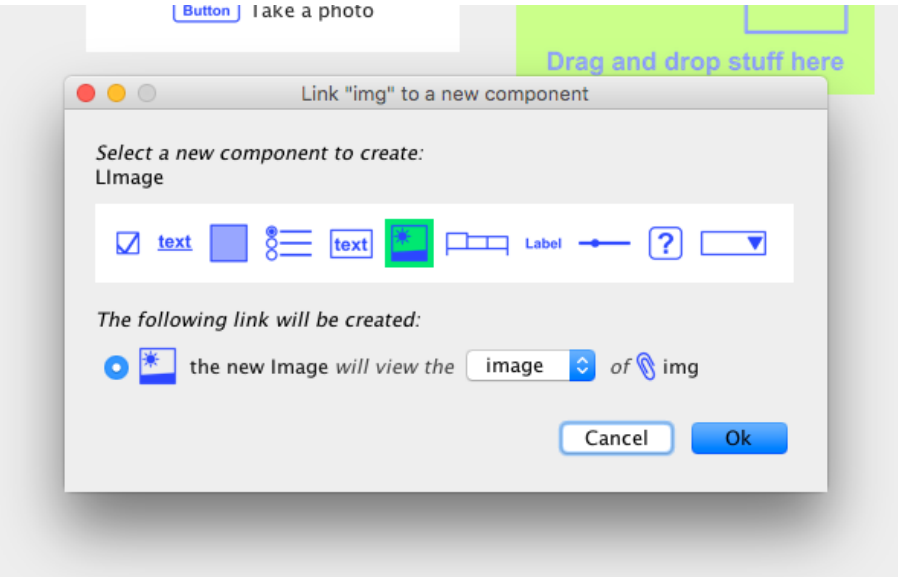


Overview of the application so far.

To view the image of the *photo*-command, we drag the image *parameter* (*img*) to the new window. It is important to note that commands need parameters to *carry information* such as text or images. The *photo*-command in itself can only be used to notify us that we got a photo from the camera, but the *actual photo* needs to be sent as a parameter.



We choose to view it as an image:

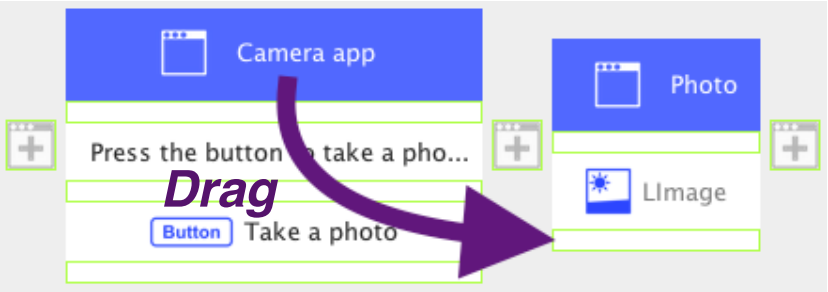


Since the image displays the *img*-parameter, we don't need to set the "image source" property in the properties dialog.

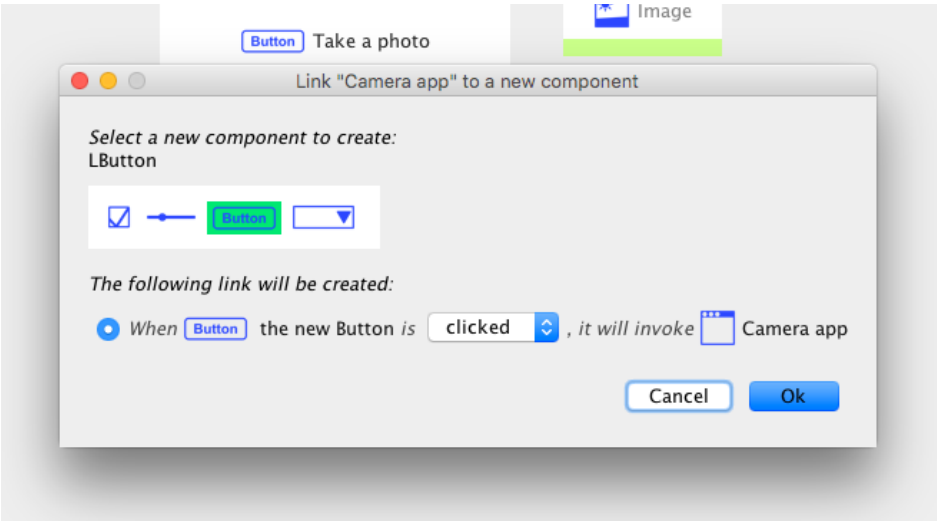
To summarize, we now have a button to take a photo and, once a photo is taken, the application will switch to a new window which displays the photo.

The last thing we need is a means to return to the original window to take another photo. We thus drag the original (“Camera app”) window and drop it below the image, and select a button (with the text “Back”) to create the link:

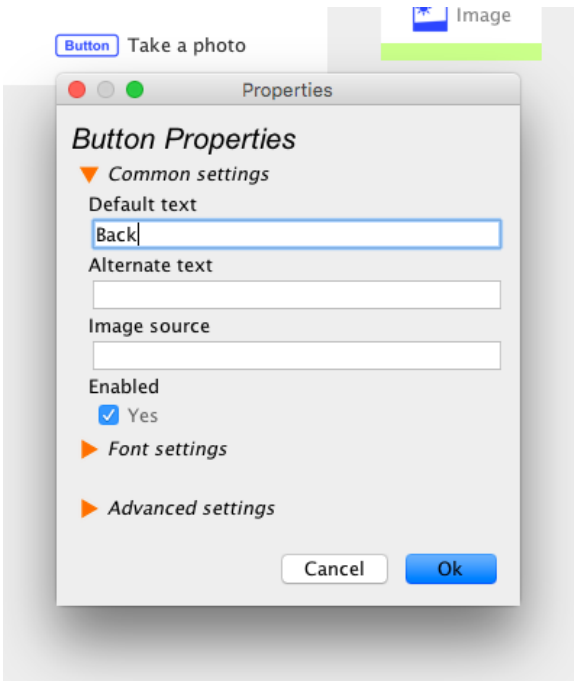
1.

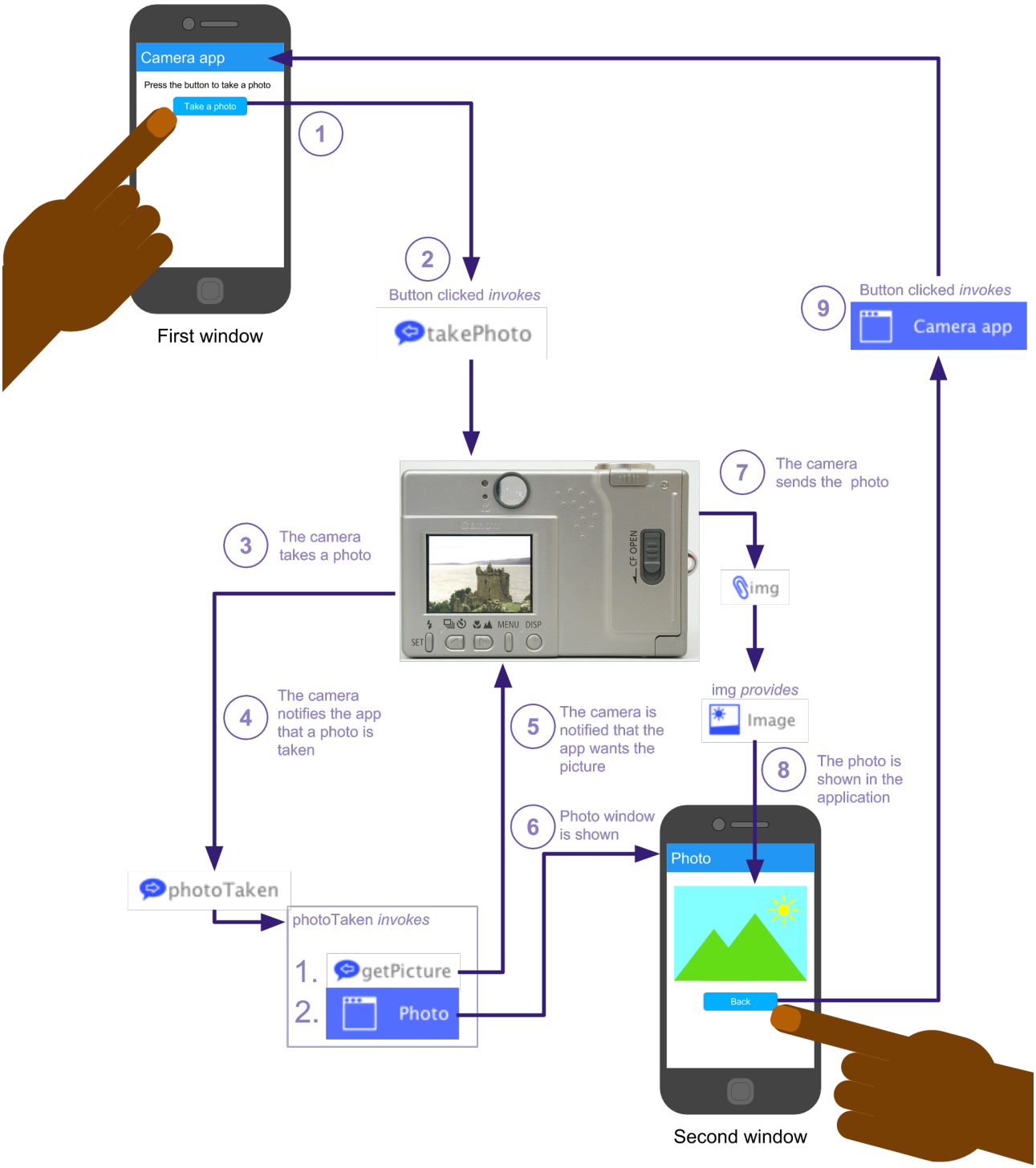


2.



3.

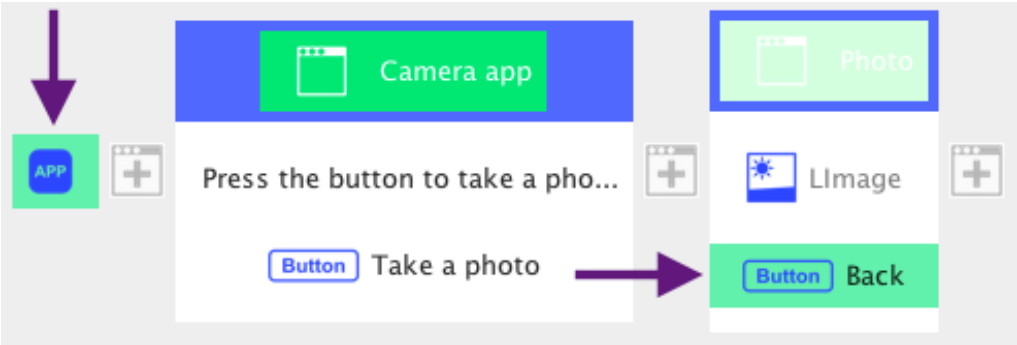




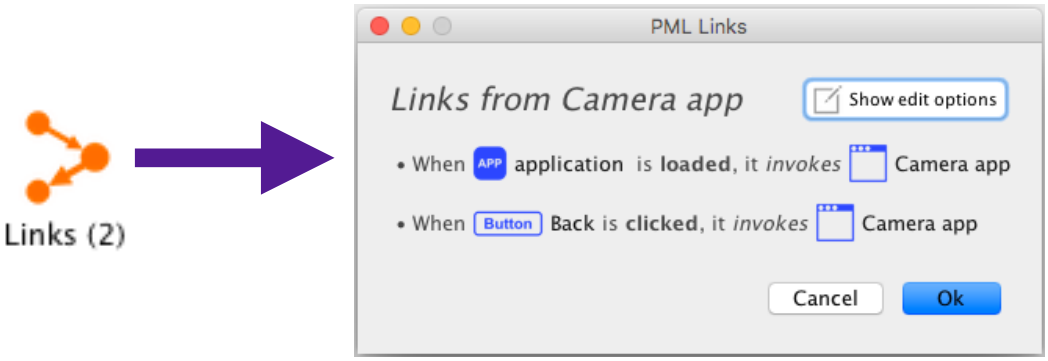
Overview of the finished application.

Now that all the functionality of the app is finished, it is a good idea to check that we haven't forgotten to link components that should be linked. The easiest way to do this is to select components, and check that the components that should be linked turn green. Remember that the parents of the linked components become light green too.

If we select the "Camera app"-window, we see that it has two links, one to the "Back"-button as expected and one we didn't create, to the "App"-component.



By clicking the "Links" button, we see that the "Camera app"-window is shown when the application is loaded. This link is added automatically, since the application wouldn't be very useful if no window was shown when it was started.



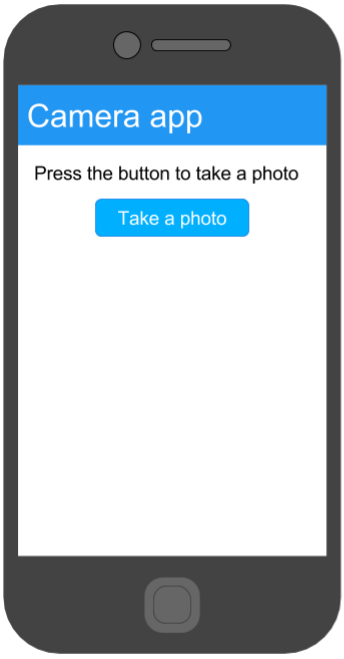
Once we have checked that all the components and links seem to be in order, we save the application by clicking the "Save"-button in the top bar. Note that the saved file can be run directly on an Android device (as well as be loaded by the editor if we want to edit it further). Once the file is saved, the save icon will turn green, indicating the file is saved.

To run the file on an Android device, you need to have an application called "AndroidPUIDI" installed. For more information about installing AndroidPUIDI, check out the *Installing AndroidPUIDI*-section.

If you have an Android device with AndroidPUIDI installed, the easiest way to run the application is to connect the device to your computer with a USB cable, and click the “Run on Android button”.

Note that the first time you click the “Run on Android”-button, you will need to locate adb (*Android Debug Bridge*) on your computer. ADB can be found free of charge on the Internet.

Now that you have followed this guide, you should have a good grasp of the basics of using the editor. Please refer to the rest of this manual for more in-depth information.



First window

Second window

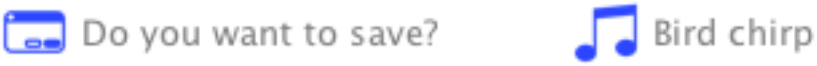
Overview

PML Components

While working in the PML editor, you will spend most of your time creating *links* between *PML components*.

PML components can be *graphical components* such as buttons and text fields:

PML components can be *notifications* such as dialogs and sounds:



PML components can be *facts* such as constants and variables:

And PML components can be *network units* such as devices and commands:

PML components all have *some things in common*.

PML components can be *linked*, i.e. one PML component can do something to the other, for example:

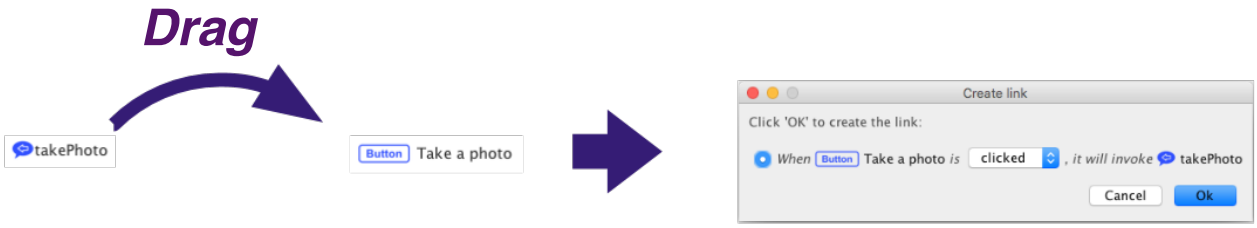
A command can be sent by a button:

A text field can supply information to a parameter:

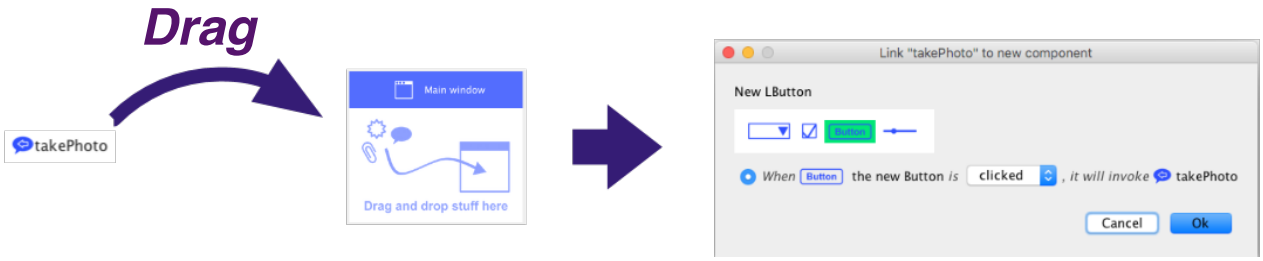
A dialog can be shown when a command is received:

Or a button can open a new window:

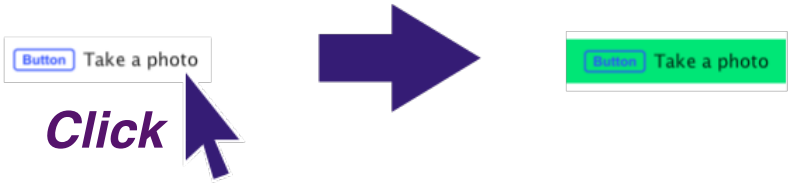
PML components can be dragged and dropped onto each other to create links:



PML components can also be dragged and dropped onto a blank spot in the canvas to link to a new PML component:



We can also select PML components by clicking on them:



When a PML component is selected, we can view and edit its links by clicking the “Links”-button:

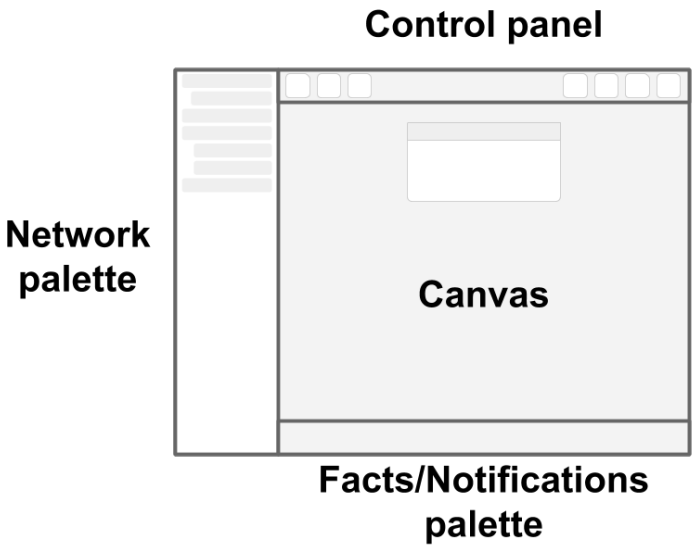
We can view and edit its properties by clicking the “Properties”-button:

And finally, when selected, we can delete it by clicking the “Delete”-button:

In the rest of the manual, we will often refer to PML Components simply as *components.*

Overview of the program

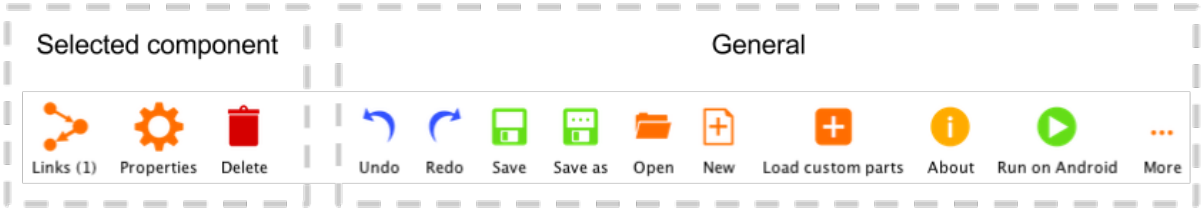
The Graphical PML Editor consists of 4 main parts:



- Control panel
  Containing options for the selected component as well as general options such as Save and Open.
- Network palette
  Where the devices on the network are listed.
- Canvas
  Which has a preview of the windows of the application, and the components in them.
- Facts/Notifications palette
  Where the facts and notifications (i.e. the components which don't have a place in a window) are listed.

The control panel

The control panel is divided into two parts, options for the currently selected component (you can select a component by clicking it) and general options.



For the selected component

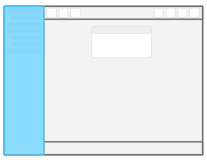
	Links	⌘/*Ctrl* + *L*	Lists the links to and from the selected component. You can delete links, or change the qualifier or order of links *from* the selected component. The number of links is shown in the button text.
	Settings	⌘/*Ctrl* + *S*	Lists the settings for the selected component, and lets you edit them.
	Delete	⌘/*Ctrl* + ⌦	Deletes the selected component.

General

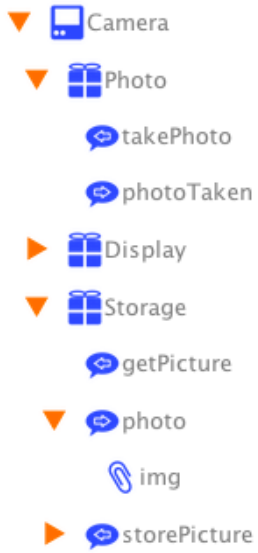
	Undo	⌘/*Ctrl* + *Z*	Undos the last action.
	Redo	⌘/*Ctrl* + ⇧ + *Z*	Redos the last undone action.
	Save	⌘/*Ctrl* + *S*	Saves the PML description. The saved description can be loaded back by the editor, or run on a PML back-end (e.g. on Android).
	Save as	⌘/*Ctrl* + ⇧ + *S*	Saves a copy of the PML description with a new name.
	Open	⌘/*Ctrl* + *O*	Open an other PML description.
	New	⌘/*Ctrl* + *N*	Creates a new PML description.
	Load custom parts	⌘/*Ctrl* + ⇧ + *L*	Load custom components into the editor, to be used in addition to the existing components.
	About	⌘/*Ctrl* + *A*	Shows info about the program, and lists any imported custom parts.
	Run on Android	⌘/*Ctrl* + *R* ⌘/*Ctrl* + ⇧ + *R*	Runs the PML Description on a connected Android device. The file will be automatically saved first. The first time you click this button, you will need to locate adb on your computer. To change the location of adb later, use the second keyboard shortcut.

•••	More		General settings for the editor.

The network palette



The network palette lists all of the units on the PalCom network. Network devices connected to the same PalCom network as the editor show up in the network palette. If you have problems with network devices not showing up in the network palette, check the *Troubleshooting* section.



Click the orange arrows to collapse/uncollapse network units that have children.

Like other PML Components, the network units can be dragged and dropped, in order to create links to new or existing components. When creating an application in the PML Editor, it is common to *begin by dragging and dropping commands and parameters to the canvas, in order to link them to new components in the application*.

For example, if you drag the takePhoto-command (in the image) to an empty spot on the canvas, you get suggestions of new components that can send the command (like a button). For more info, see the *Creating components* and *Links* sections.

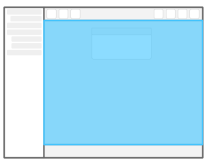
An important difference between commands and parameters is that parameters can *carry information*, like text and images. Commands on the other hand can only be sent or received, and can thus only be linked to components that can either send the command or react to the command being received. To send for example the text of a text field to a device, we need to use a parameter.

List of network units

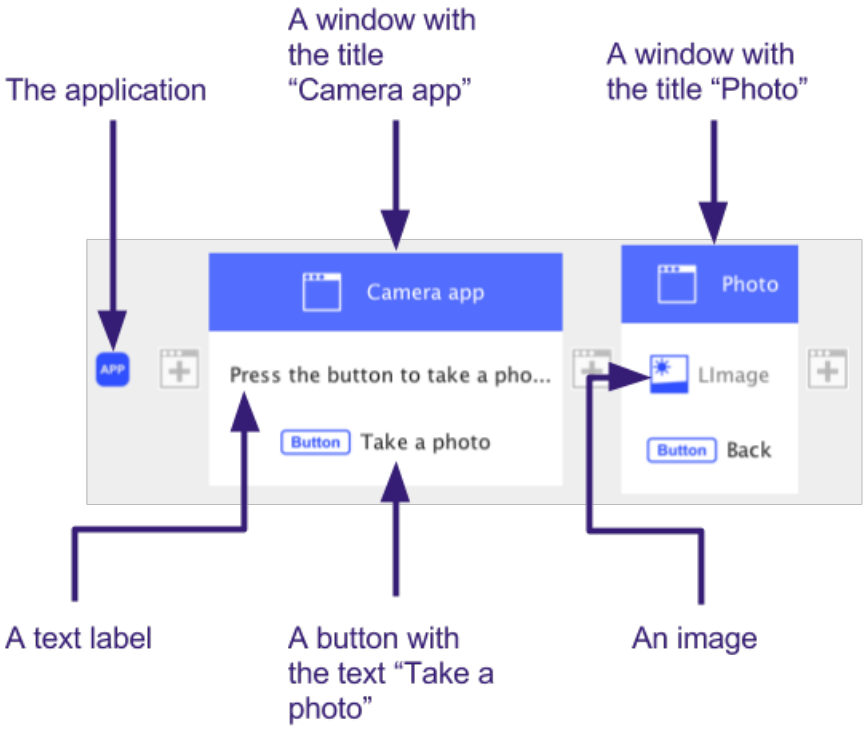
Network units

	PalCom Device	A device on the network, which may contain services.
	Service	A service represents something a device can do. Services may contain commands. By linking a service to other components, it can make things happen when it turns available or unavailable.
	Command *to* the device	A command to a device. It can be sent from the application by a component, for example a button.
	Command *from* the device	A command from the device to the application. It can make something happen in the application, for example show a dialog.
	Parameter	Commands can have parameters, carrying text or images to and from a device. Text for example can be supplied to a parameter by a text field (for commands *to* the device) or be viewed by a label (for commands *from* the device)
	New static component	Sometimes, you might want to create components without links (at least initially), for example text labels that should have the same text all the time, or new areas for structuring your application. Then you can drag the ‘New static component’-label to where you want to create the component.

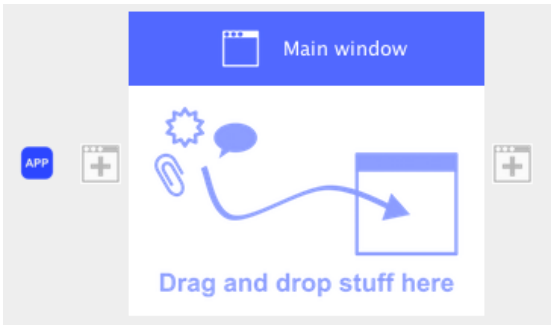
The canvas



The canvas shows a preview of what the finished application will look like:



When starting a new file, the canvas will look like this:



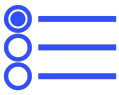
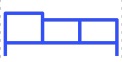
The new file contains an empty window with the title “Main window”, and a small “App” component:

Links between components are created by dragging and dropping the components we want to link, and thus the “App” component is used to create links with the application itself, e.g. to make something happen when the application is *loaded* or *unloaded*. A link between the application and the main window is created automatically, which shows the main window when loading the application.

Beside the main window are two “Add window” icons, which are used to create new windows:

List of components in the canvas

Components in the canvas

	The Application	The component that *represents the application we are creating*. By linking to this component, we can make things happen when the application is loaded or unloaded.
	Area	The *main layout component*, which can contain components and arrange them using its layout settings. The area can also be hidden/shown (by linking to other components giving it the value 0 or 1). To move or link an area, click between its components to select it, and drag and drop the crossed arrows-symbol that appears.
	Button	A button that can make things happen when its clicked. Furthermore, it can take and provide pictures as well as open a file dialog and provide the chosen file path. It can also be disabled/enabled (by linking to components giving it the value 0 or 1).
	Checkbox	A checkbox that can make things happen when it's selected/unselected, and can get its value from another component as well as provide its value to another component.
	Custom part	Custom parts can be created and loaded into the editor, to expand its capabilities. What custom parts do and what they can link to is up to the developer. Custom parts can have their own icons in the editor.
	Dropdown list	A dropdown list, which the user can click to view a couple of selectable options. It can provide its selected value to other components as well as get its selected value, selected index, or content from other components. It can also make something happen when a value is selected by the user.
	Image	An image, that can both provide its image to another component as well as get its image (or mask) from another component. It can be set to be editable, enabling the user to draw on it.
	Label	A label containing a text which can't be edited by the user. It can get its text or font color from another component, and can be hidden or shown (by linking to a component giving it the value 0 or 1). Labels with no text can be used to create spacing in grid layouts.
	Number slider	A slider which the user can use to select a value between a specified minimum and maximum value. It can get its value from another component or provide its value to other components. It can also make something happen when a value is selected by the user.
	Radio group	A container for radio buttons. It can provide its value as well as get its value by linking to other components. It can also be hidden/shown (by linking to a component giving it the value 0 or 1). New radio buttons can be created by clicking the small grey "+"-symbols in the radio group.
	Radio button	A radio button, with a value and a text. Radio buttons need to be in a radio group. The text is shown to the user, and the value is sent/received to other components.
	Tabbed	Tabs are used to structure the application. Adding tabs is done in the Properties-dialog of the tabs. A tab can be selected by another component, and the tabs can be disabled/enabled by another component (if it gives the tabs the value 0 or 1).
	Text area	The text area functions the same way the label does, except that the text area allows for multiple rows of text.
	Text field	A text field which the user can edit. By linking it, it can provide its text to other components or get its text from other components. It can also be disabled/enabled by linking it to a component giving it the value 0 or 1 (with the enabled qualifier).

	Window	A window, that can be opened by other components (e.g. by the application or by a command). It can also make something happen when it's opened or closed. Like the area, the window contains other components, and can have different layouts. Only one window in the application will be displayed at a time.

The facts and notifications panel

Facts and notifications are components in the application that don't have a specific position in the canvas. Notifications can be used to notify the user of something or prompting the user to make a choice immediately, typically by showing dialogs. Facts can be used to update components with constant values (e.g. reset a text field or a list with a default value), keep track of values in the application or create links between components that can't be linked directly. For more information of the things that can be done with facts, check out the *Using facts*-section.

List of notifications

Notifications

	Quick note	Displays a pop-up which displays a text to the user. We can link other components to make it show.
	Sound	Plays a sound. We can link other components to make it start playing, stop playing, or toggle between whether it's playing or not.
	System notification	In android, this will give a system notification, i.e. show up with the notifications from other applications. We can link other components to post it.
	Yes/No-dialog	The yes/no-dialog is a versatile dialog. It prompts the user to choose between two choices (typically yes or no, but this can be customized). It can then make different things happen depending on which option the user chooses. By linking to other components, we can make the dialog show or disappear. Additionally, if we set the dialog to provide text to another component, the dialog will include a text field where the user can input text.

All notifications can be disabled/enabled by linking them to components which gives them the value 0 or 1 (for the enabled qualifier).

List of facts

Facts

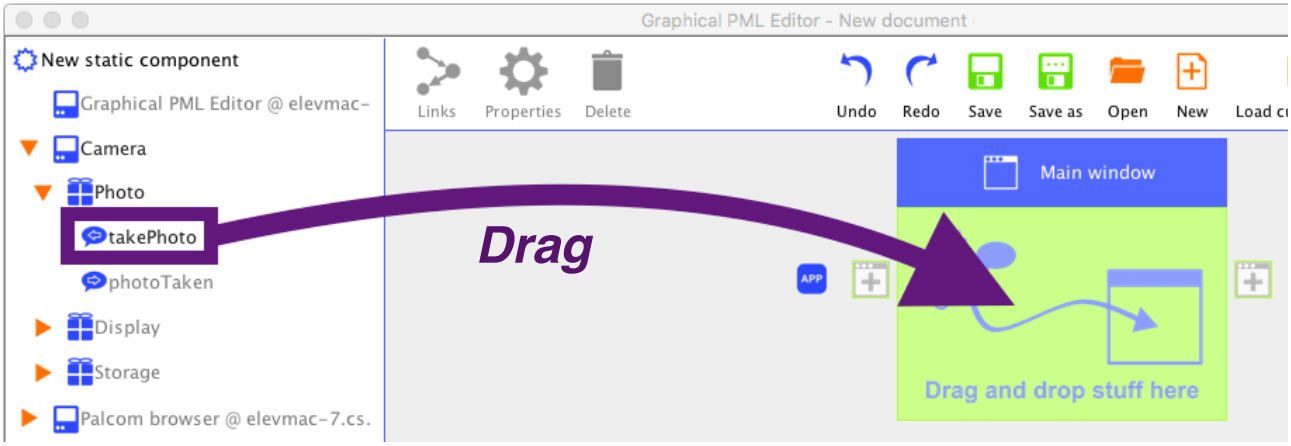
	Constant	A constant value, that can be provided to other components. The constant will provide its value when it is invoked by another component (e.g. a button can be clicked to send the value of the constant to the components it provides it to).
	Variable	A constant with the added functionality of being able to view the value of other components, i.e. its value can be changed. While a constant needs to be invoked to share its value, the variable can be set to auto-invoke, which means it will provide its value whenever it gets a new value.

Creating components

Most of the components in the GUI can be dragged and dropped to be moved or to create links to new or existing components.

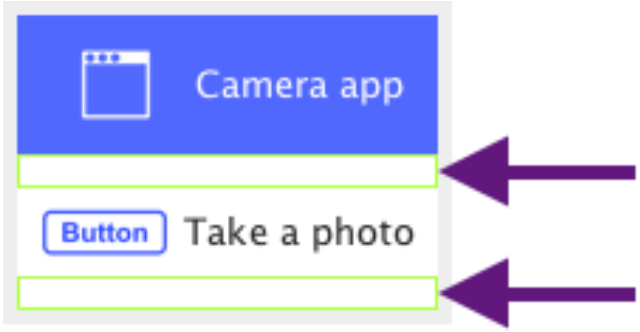
Creating components with links

To create a new component with a link, just drag the component you want to link to an empty slot in the canvas.

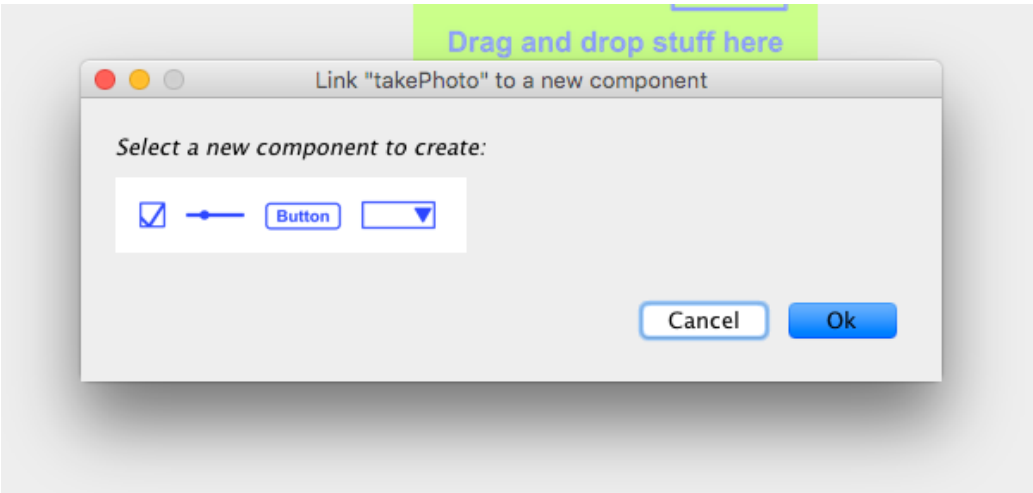


In an empty window, the empty slot to which components can be dropped looks like this:

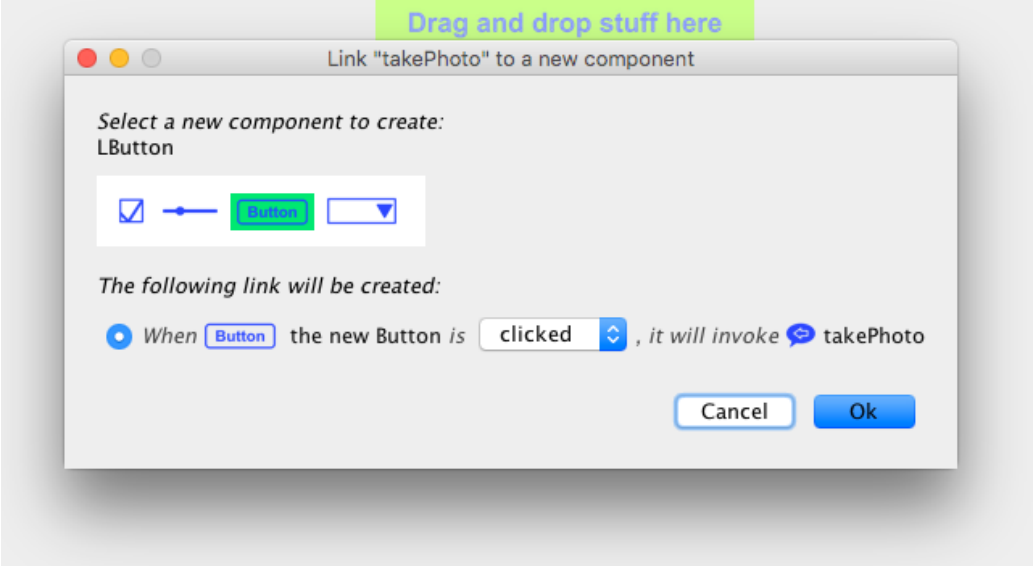
If a window already has components, components can be dropped next to the existing components:



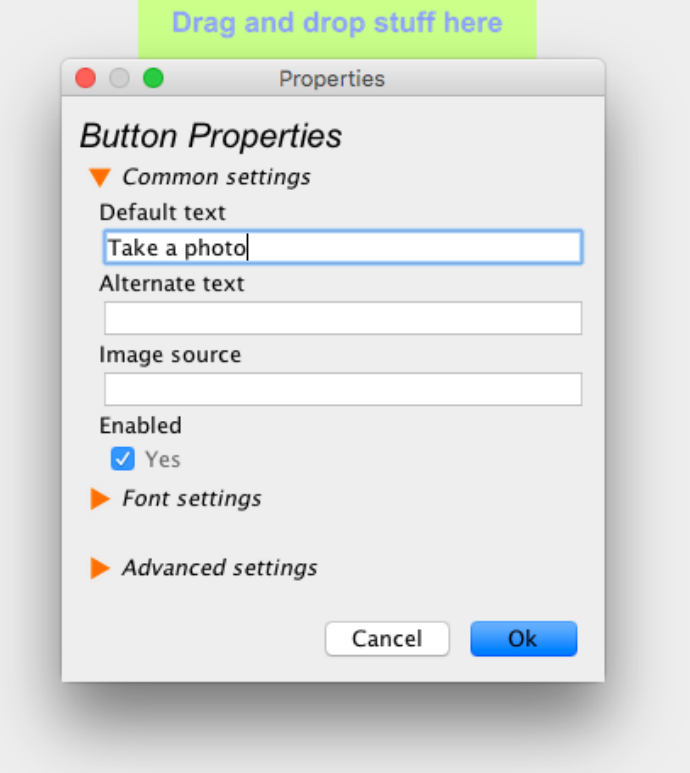
Dropping a component you want to link to a new component in an empty slot on the canvas brings up a dialog with choices for new components that can be linked to the component that was dropped:



When selecting one of them, the link that will be created is shown:



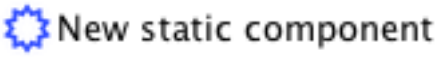
After having selected a component and link to create, properties for that component can be set. Clicking OK creates the component.

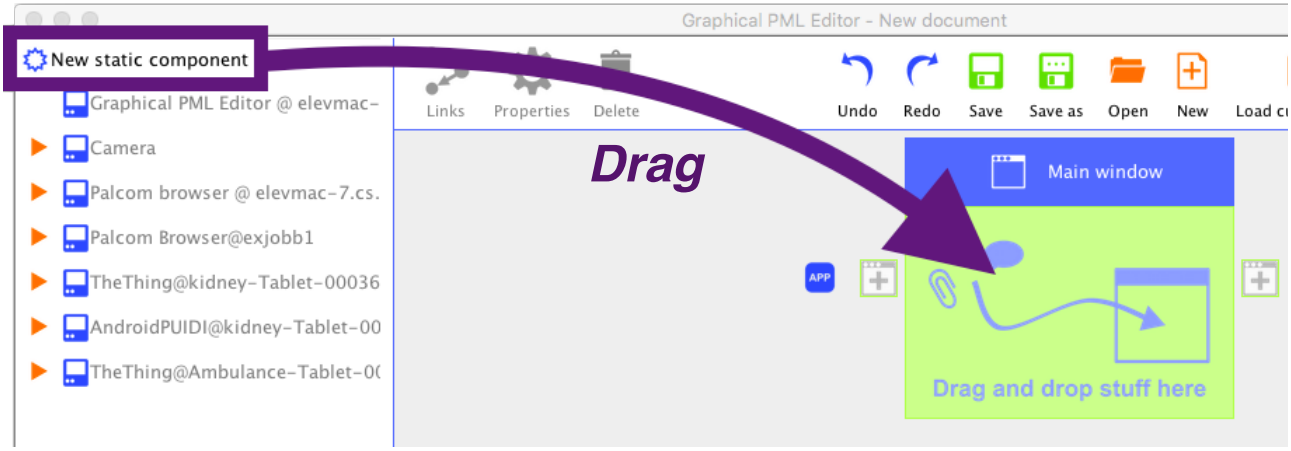


Creating static components

Sometimes you might want to create components that aren't linked to anything, at least initially. For example, you might want to have a text in the application that doesn't change.

Creating static components is done similarly to creating linked components, except that instead of dragging the component you want to link, you drag the "New static component"-label to where you want to create your new component.

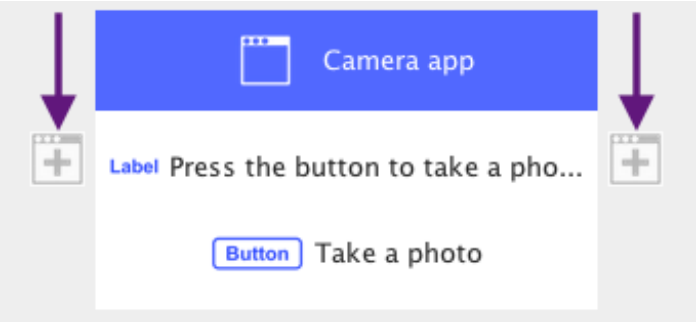




Even though static components don't get any links when created, you can add links to them later (by dragging and dropping components you want to link onto it). Be wary that many components without links don't do anything (a button without a link won't do anything when pressed, for example).

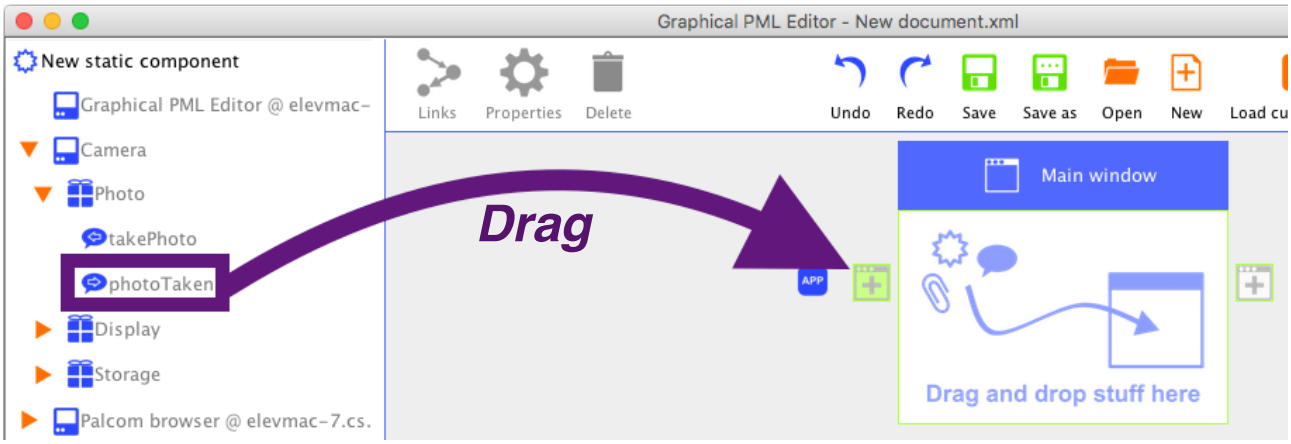
Creating windows

Since windows can't be created *within* other windows, they need to be created at the small "Add window"-icons, located between the windows:



To create a new static window, you can just click the "Add window"-icon:

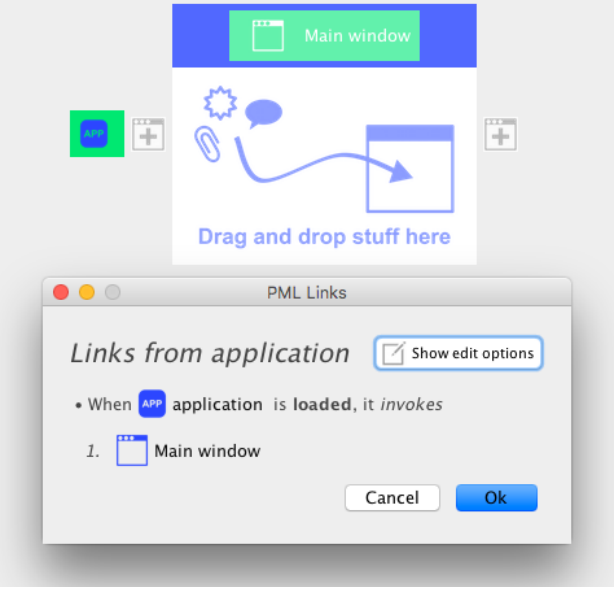
Creating new windows with a link is done similarly to creating other components, except that you drag the component you want to link to one of the "Add window"-icons:



The links that are available for windows are:

- Something can *invoke* the window, e.g. a button can show the window
- Something can *be invoked* when a window is either opened or closed.

Note that the order of windows in the editor don't matter. A window must have a link to something that invokes it in order to be shown. When creating a new file, a link between the application and the first window is created automatically, which shows the first window when the application is started:

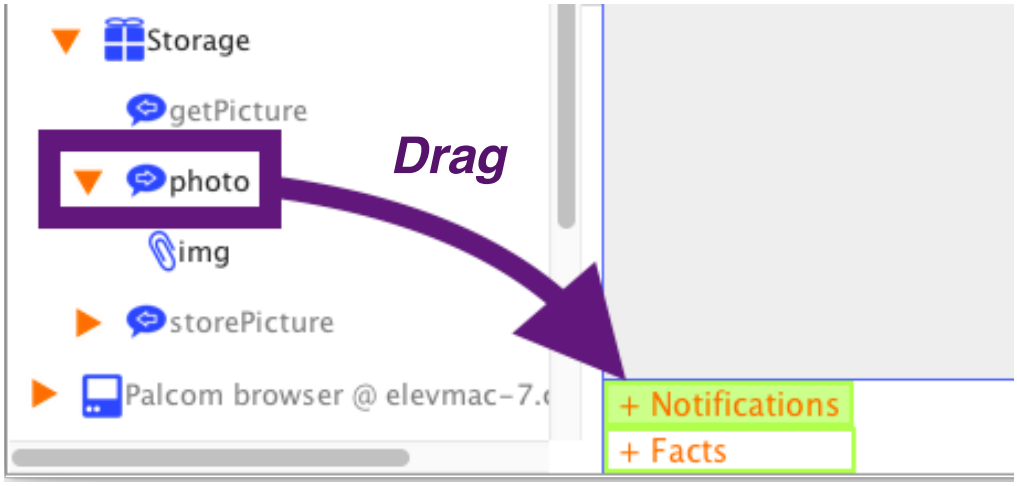


Creating notifications and facts

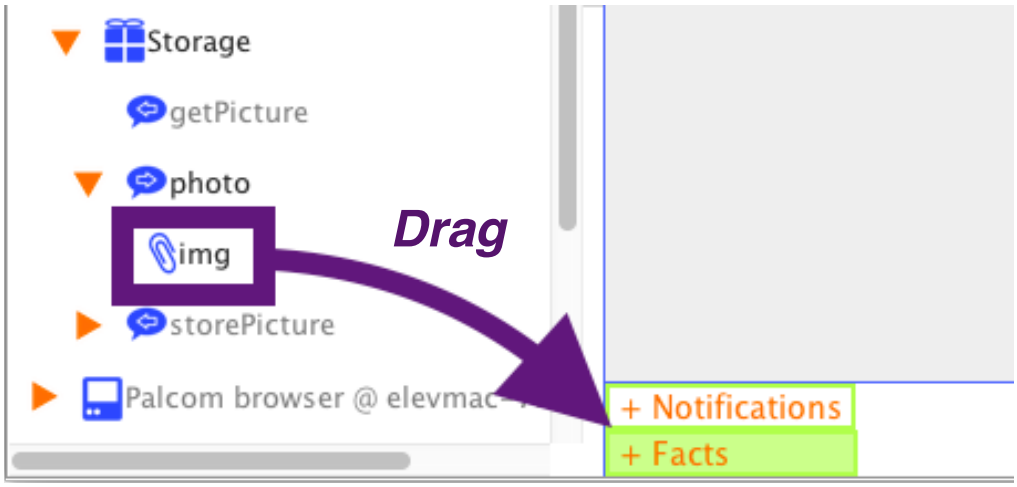
Notifications are components like pop-up dialogs and system notifications. Facts include constants and variables.

Similarly to windows, notifications and facts can't be created in windows like most other components. To create new notifications or facts, you can simply click the orange "Notifications" and "Facts" labels in the facts and notifications panel.

To create a notification with a link, just drag the component you want to link to the "Notifications"-label. Note that a notification need a link to be shown (or in the case of Sound, to be played).



To create a fact with a link, just drag the component you want to link to the "Facts"-label. Both constants and variables can provide their value to something through a link, but only variables can have their value changed through a link.



Links

What is a link?

Generally speaking, links specify the *functionality* of the application, i.e. links *make something happen* in the application. In order for a button to do something for example, you need to link it to another component. Correspondingly, in order for the text of a text field to be used for anything, you need to link the text field to another component.

Some examples of links are:

- When a *button* is clicked, send a *command*
- A *parameter* should contain the text of a *text field*
- When the *application* is loaded, show a *window*
- A *parameter* should supply a *dropdown list* with its selected value.

Link types

There are four basic link types:

1. *invoker*
2. *provider*
3. *reactor*
4. *viewer*

Invoker links are a way for one component to *tell another component to do something*. An example of an invoker link is when a button is used to send a command. We can think of it as the button "poking" the command and saying "Do your thing now".

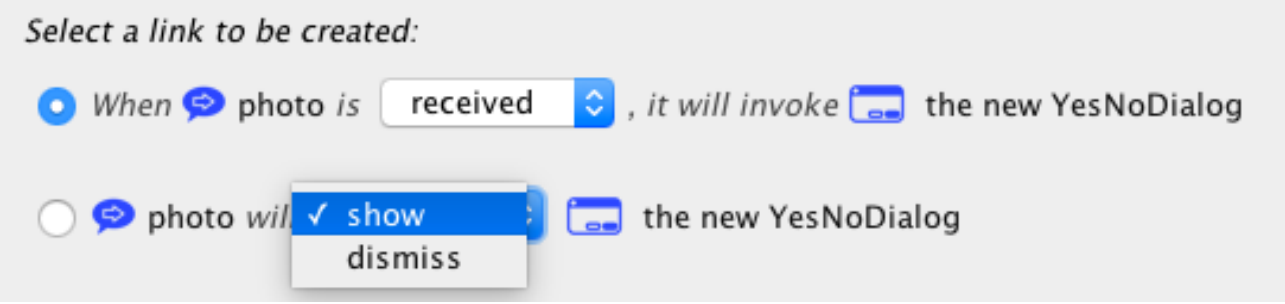
Provider links on the other hand are a way for one component to *supply information to another component*. An example of a provider link is when a text field *provides its text* to a parameter.

One of the components in a link have a qualifier. Qualifiers provide more fine-grained control of the links, like what event should trigger the invoker link, or what should be provided in a provider link. Examples of qualifiers in links are:

- When the user *chooses "No"* in a yes/no-dialog, send a command (Choosing "No" is the qualifier).
- A parameter provides a dropdown-list with its *selected value* ("selected value" is the qualifier).
- When a button is *clicked*, send a command ("clicked" is the qualifier).

Sometimes, there's only one possible qualifier for a link (a button can only send a command when clicked) which renders the qualifier something of a formality.

A r*eactor link* is basically the same as the invoker link, except that *the component that reacts in the link has the qualifier*. We can for example link a *command* from a device to a *yes/no-dialog* in *two ways*: with an invoker link and with a reactor link:



The first link in the image above is an invoker link: the command "photo" *invokes* the yes/no-dialog. The command thus has the qualifier: *received*. The second link is a reactor link, since *the component that reacts* (the yes/no dialog) has the qualifier. The qualifier can be either to *show* or to *dismiss* the yes/no-dialog when the command is received.

The component that doesn't have a qualifier will behave according to its default behavior: the yes/no-dialog will *show* in the invoker link, and the command will make the yes/no-dialog show/dismiss *when received* in the reactor link. A list of the default behaviors can be found in the *List of Links*-section. In that section, the component with the qualifier in a link is referred to as the *"From"-component*, and the component without a qualifier is referred to as the *"To"-component*.

One last thing to note about the difference between the "From"-component (the component with the qualifier) and the "To"-component (the component without a qualifier) in a link, is that the "From"-component can define the order in which the links happen. If the *photo*-command in the example above has more than one invoker link, we can decide *which order the linked components will be invoked* when the command is received. More information about editing the order of links can be found in the *Editing and removing existing links*-section.

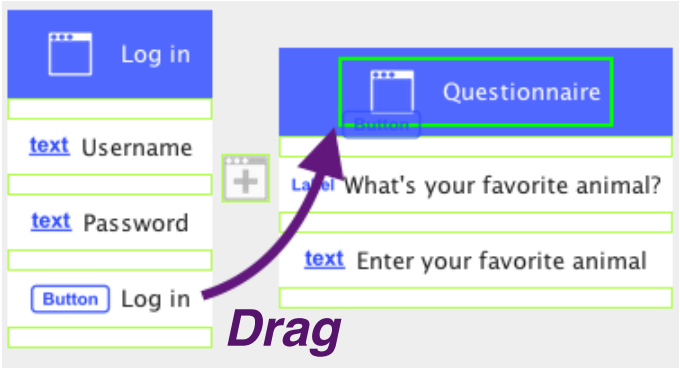
Lastly, we have the *viewer* link type. Viewer links are to provider links what reactor links are to invoker links, i.e. in a provider link the component that *supplies information* has the qualifier, and in a viewer link the component that *receives information* has the qualifier. A dropdown-list for example, can view information from another component as its *content*, its *selected value* or its *selected index*, i.e. a viewer link with the drop-down list can have three different qualifiers.

For advanced users, in case you really need to have qualifiers for both components in a provider/viewer-link, you may use a *variable* as a middle-man. This can be done by creating a provider link between the first component and the variable, and a viewer link between the variable and the second component. Make sure the *auto-invoke* property of the variable is on, or you will need to invoke the variable to make it give its value in the viewer link.

Creating links between existing components

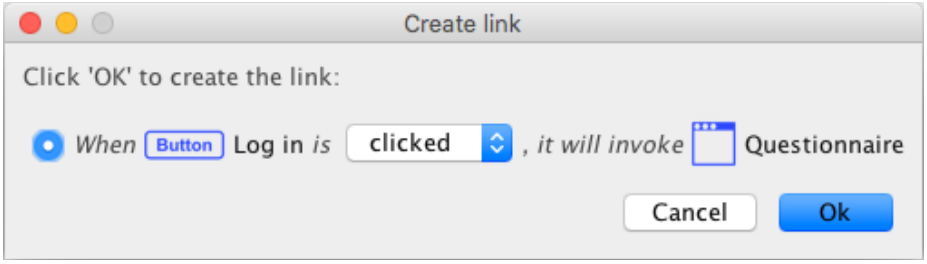
Creating a link between two components is done by dragging the first component and dropping it on the second component. If a link is possible, the border of the second component turns green, otherwise it turns red.

A button, for example, can be used to switch over to another window, and thus a link is possible. Since a link is possible, the window will get a green border when dragging a button over it:



A button and a text field on the other hand, can't be linked. The text field thus gets a red border when dragging a button over it:

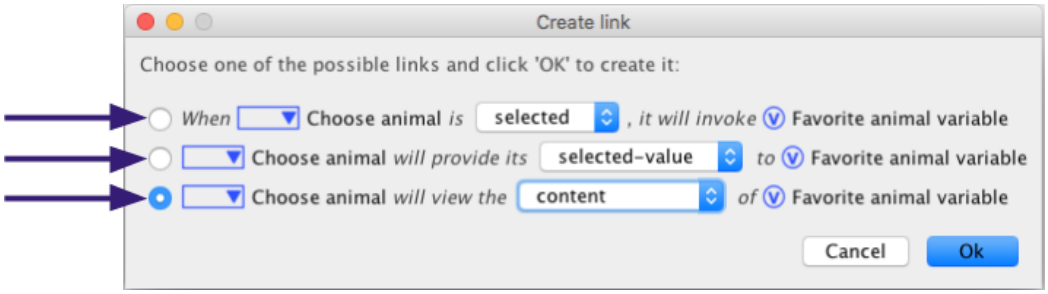
After dropping a component onto another where a link is possible, we get a dialog explaining what the link that we are about to create will do. We can create it by clicking OK:



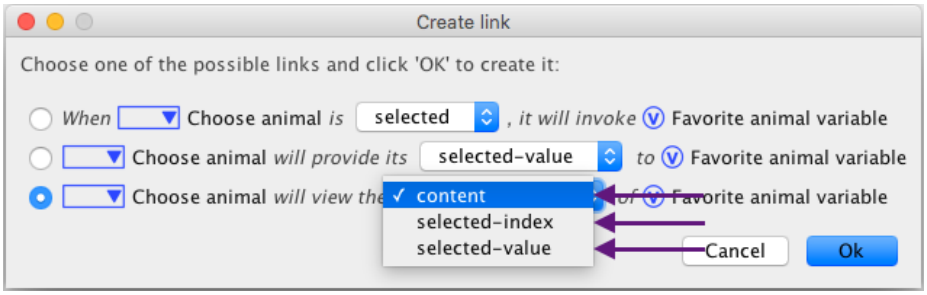
In some cases, more than one link is possible between two components. A dropdown-list for example can have three different basic kinds of links to a variable:

5. When the dropdown-list is selected (someone is changing the value), it can *invoke* the variable, (make the variable give its value to any of its linked components that views its value).
6. The dropdown list can *provide* its selected value to the variable (e.g. “2” if option #2 is selected).
7. It can *view* the content of the variable, e.g. if the variable has the value “cat,dog,giraffe”, the dropdown list will have the options “cat”, “dog” and “giraffe”.

The first option is an invoker link, the second option a provider link, and the third a viewer link. When dragging and dropping a dropdown-list (called *Choose animal*) onto a variable (called *Favorite animal variable*), we can choose which link to create in the dialog that pops up, by clicking one of the radio buttons:



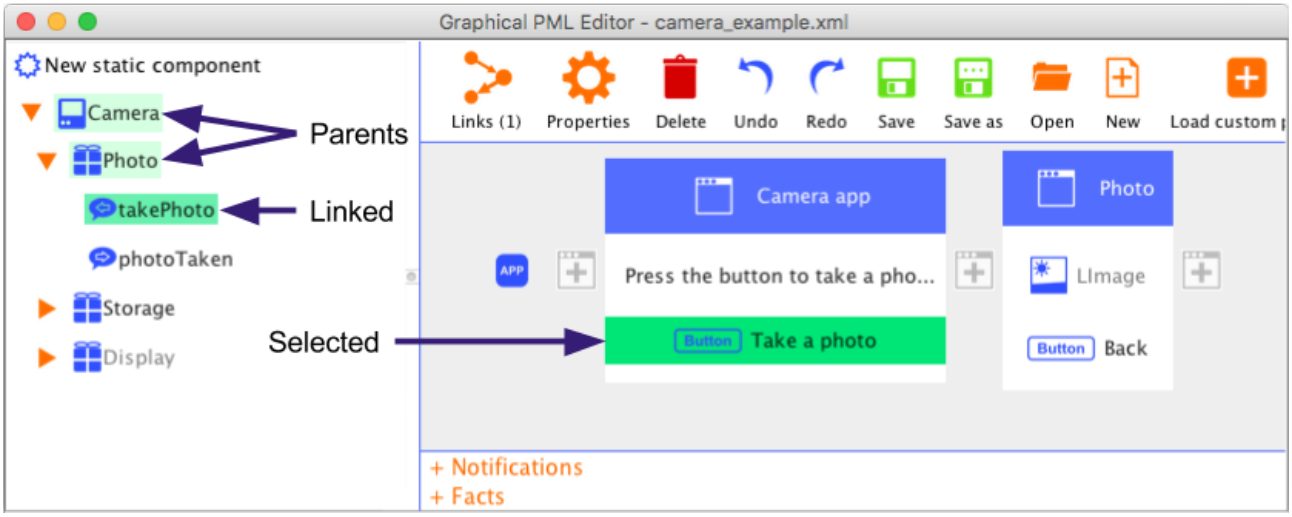
In some cases, more than one qualifier for a link is possible. Then we can choose which one we want in the dropdown dialog in the link description.



Getting an overview of links

The links between components define the *functionality* of the application, i.e. how the application behaves and what the application can do. Getting an overview of the links in an application is thus important to understand how the application works. In addition, getting an overview of the links helps us spot missing functionality, as well as parts of the application that doesn't behave the way we want.

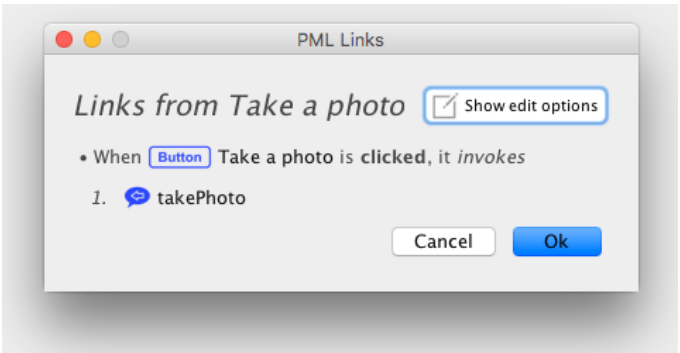
The fastest way to get a quick overview of the links to and from a component is to select the component (by clicking it). When a component is selected, it becomes bright green, but any linked components become green as well. Any parent components to linked components (windows, parent network units…) become light green, to make it easier to locate linked components.



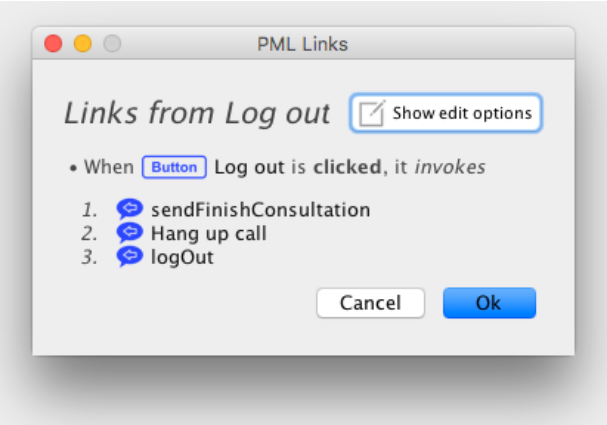
This is especially convenient when trying to grasp the workings of a large application, or to confirm that we didn't forget to create any links between components that should be linked.

In addition to the coloring of linked components, the "Links"-button will display the total number of links to and from the selected components:

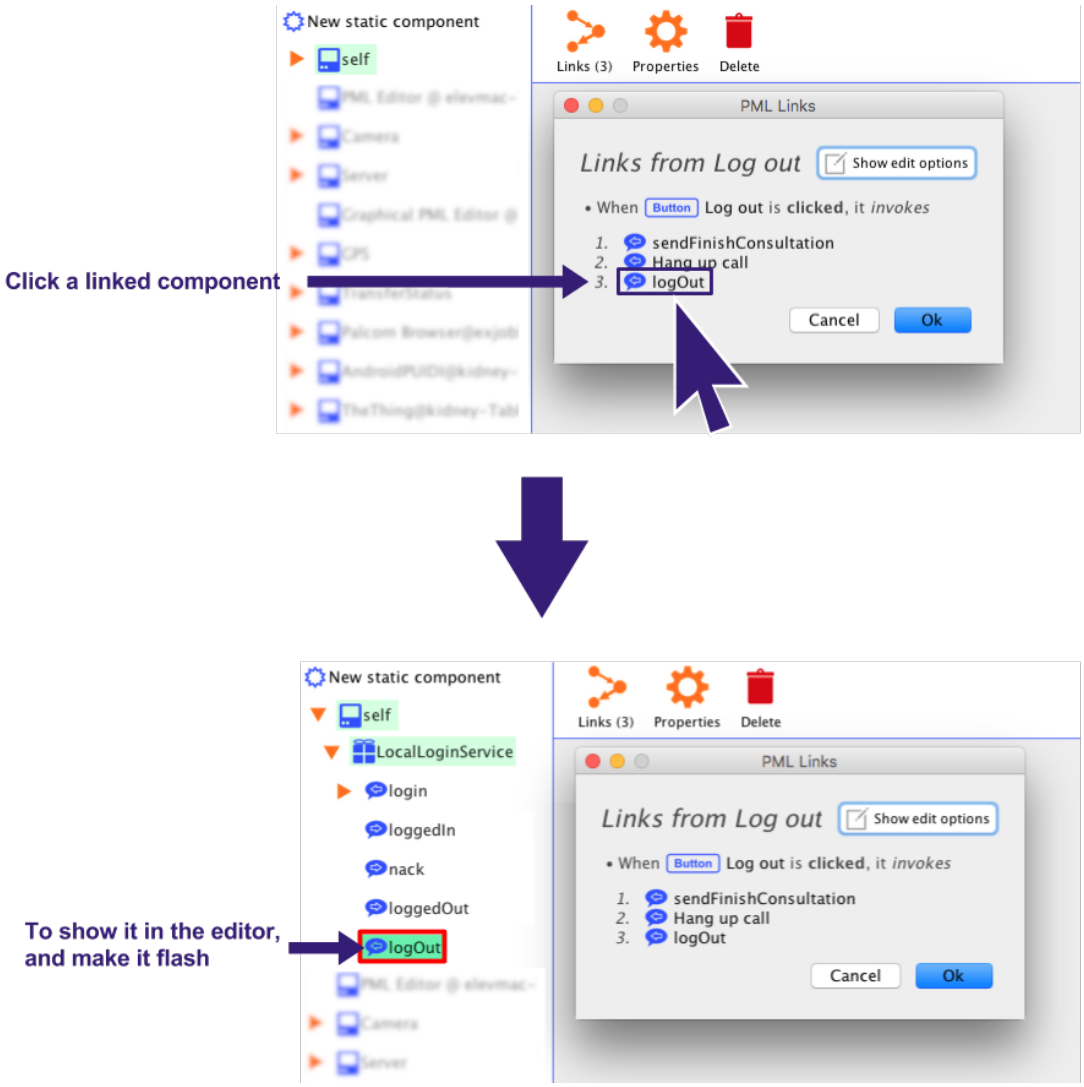
When clicking the "Links"-button, a more detailed listing of the links to and from the component will appear:



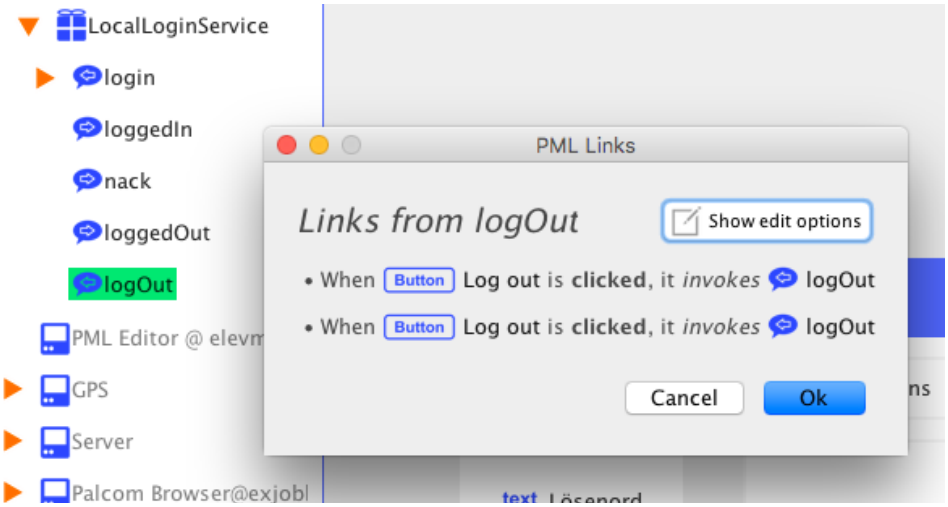
If a component has more than one links *from* itself, the order of the links will appear. In the image below, the links for a “Log out”-button are listed. When the button is clicked, three commands will be sent. *Firstly*, the command called “sendFinishConsultation” is sent, *secondly* the command called “Hang up call” is sent, and *finally*, the command called “logOut” is sent.



If you are uncertain which components are which when viewing links, you can click them to make them show and flash in the editor. Let’s say we want to know which service the “logOut”-command belongs to. Then we can click the “logOut”-command in the “Links”-dialog to make it show in the editor. We then see that it belongs to “LocalLoginService”.



If we then want to know what links the “logOut”-command has (in addition to the “Log out”-button), we can select it and click the “Links”-button again (after having closed the current “Links”-dialog). As it turns out, the “logOut”-command is sent by two buttons with identical names:

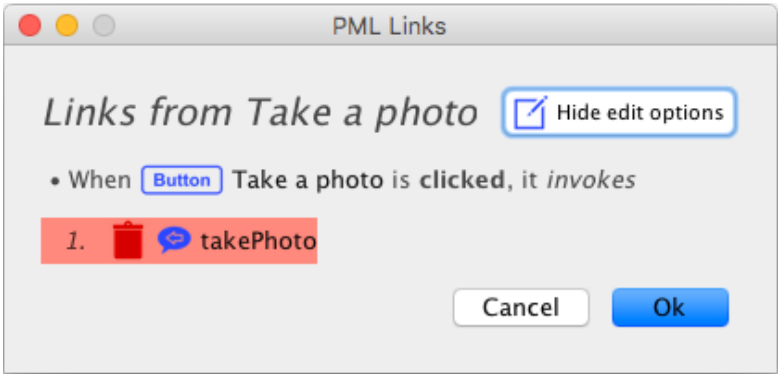


To know which is which, we can click them in the dialog, which will make them show in the editor. Clicking them like this is thus both handy for locating linked components as well as differentiating between components with identical names.

Editing and removing existing links

To edit links, first select a component (by clicking it), and then click the “Links”-button. This brings up the Links dialog. Click the “Show edit options”-button.

This brings up some more options. All links have the option of deletion: click the trash can icon to mark the link to be deleted:



When we click OK, all the links marked to be deleted will be deleted.

Links *from* the selected component may have more options:

- Some links can have many possible *qualifiers*. A yes/no-dialog for example can invoke different links depending on if the “yes” or “no” button is clicked, and a drop-down dialog can view a text as both its selected index and its selected value. If a link has many possible qualifiers, you get the option of changing the qualifier.
- If the component has more than one link for a specific qualifier, we get the option of changing the order of the links. The order of the links is the order in which the links happen.

The qualifier can be changed by clicking the "…"-button next to a component. Note that this option only appears for links *from* the component, and when *more than one qualifier is available:*

The order of links can be changed by clicking the up and down arrows next to the component. Note that this option also only appears for links *from* the component, and when *more than one link exist for the given qualifier*:

Below, we will illustrate these two options with an example: A dropdown list called "Choose your country". This example is from a form to create an account, where the user has to specify her country of residence with the dropdown list. You may skim the example if you want.

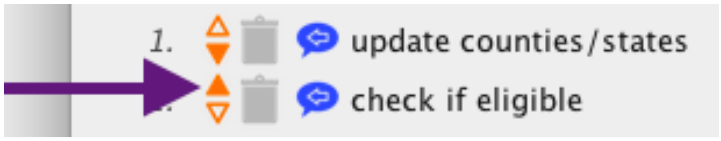
After having selected a country in the dropdown list, the *update counties/states* and *check if eligible*-commands will be sent in that order. The *update counties/states*-command is sent to a service which then updates another dropdown list with counties/states for the selected country (so

that the user can input her county/state), and the *check if eligible*-command is sent to a service which decides if a user from the supplied country is eligible to create an account. Both of these commands have parameters which get their value from the *Choose your country*-dropdown list. In addition, the selected country is supplied to a *create user*-parameter (which belongs to a command to create a user, which is sent later when all the information has been filled in).

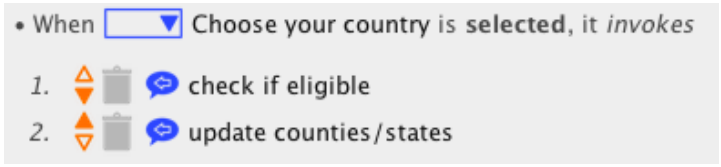
In addition to this, the dropdown list views the *selected index* of a *reset list*-command, which has the value 1. This constant can be invoked to reset the list back to its first option. The dropdown-list also views the *selected value* of a *previously entered country*-parameter, which is sent if the user has previously entered her country of residence (or a language that implies a country) so this option is pre-selected.

Let's say we want to change the order in which the dropdown-list sends the commands (*update counties/states* and *check if eligible*) when a country has been selected: It might be better to check if the country is even eligible for creating an account before we suggest counties/states for that country. To change the order in which the commands are sent, we click the "Show edit options"-button. This views the edit options for the dropdown-list:

To move the *check if eligible*-command one step up, we click the orange up-arrow next to the command (or the orange down-arrow next to *update counties/states*):

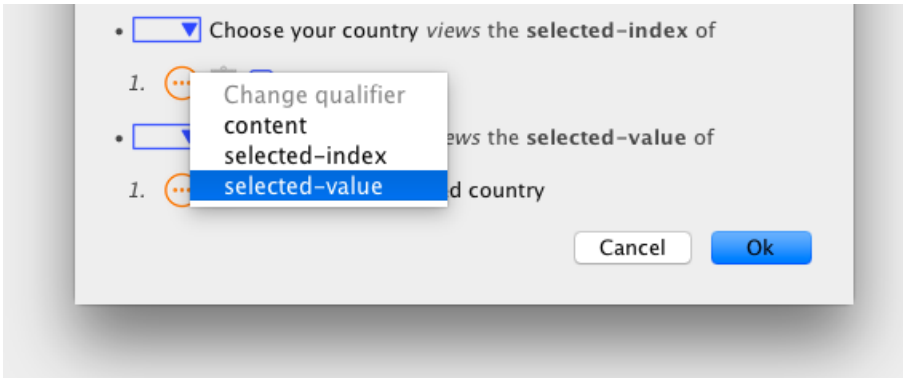


We then see that the order has been changed, and that *check if eligible* will be sent before *update counties/states*.

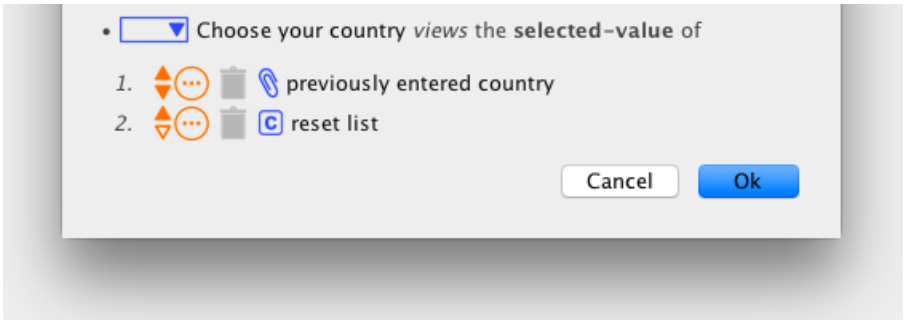


Let's say we want to change the *qualifier* for the *reset list*-constant. We might realize that the majority of our users are from the United States, and thus want the US to be the value to reset the list to (and not Andorra, which is at index 1). The easiest way to do this is to change the qualifier from *selected-index* (which is a number) to *selected-value* (which is the name of the option), and change the value of the *reset list*-constant to "United States" later. To change the qualifier, we click on the orange "…"-button next to the *reset list*-variable:

This brings up a list of the possible qualifiers, i.e. the ways the dropdown list can view the constant. We click *selected-value* to change it to that:



We then see that the *reset list*-constant has been moved to the *selected-value*-qualifier:



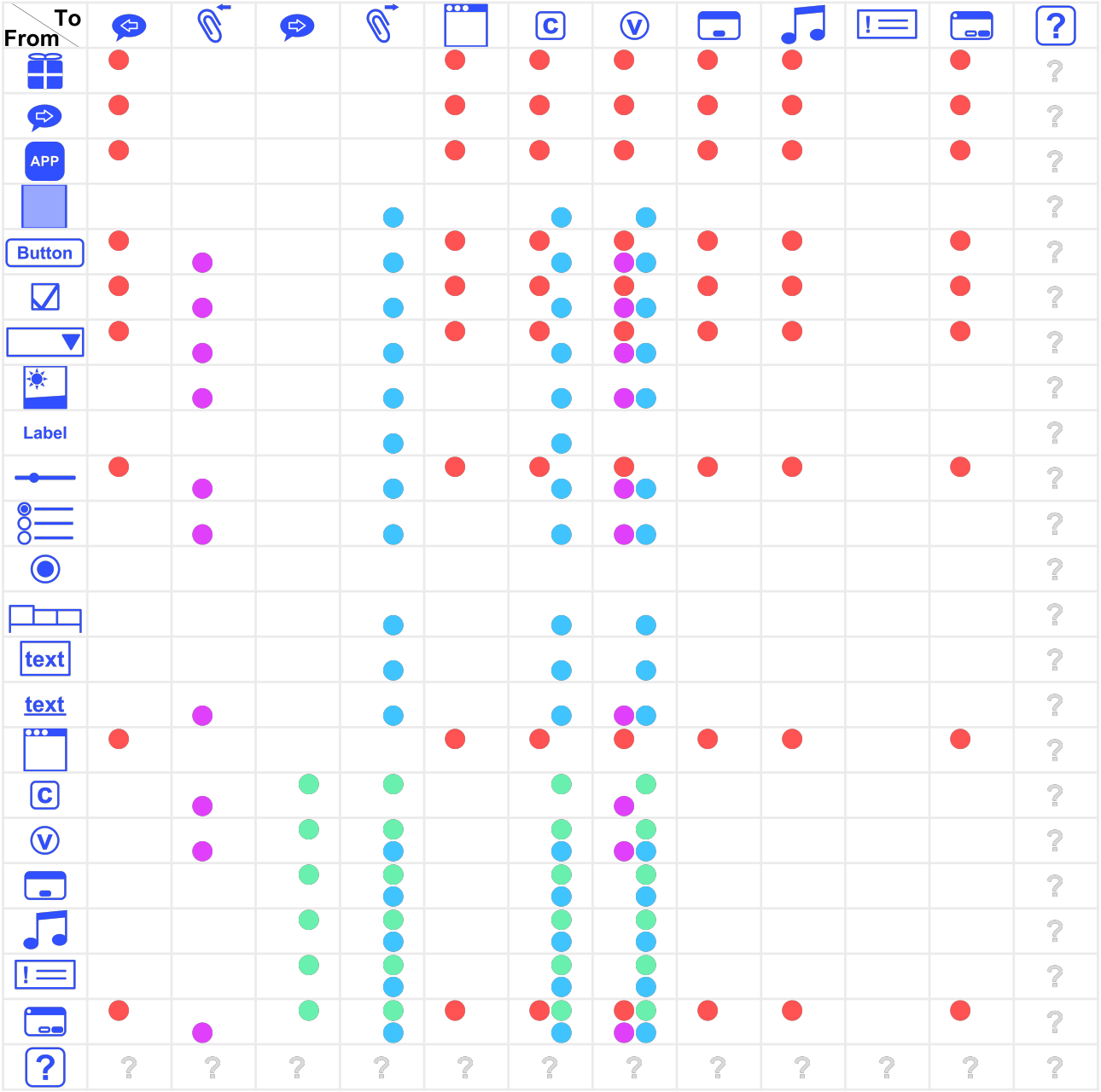
We click OK to save the changes (and proceed to change the value of *reset list* to “United States”).

List of links

Link types

Symbol	*Link type*	*Explanation*	*Example*
	Invoker	The *"From"*-component can *invoke* the *"To"*-component.	The application can *show* a yes/no-dialog when loaded or unloaded.
	Reactor	The *"From"*-component can *react to* the *"To"*-component.	A yes/no-dialog can *be shown or dismissed* by a command from a device.
	Provider	The "From"-component can provide something to the "To"-component.	A text field can *provide its text* to a parameter to a device.
	Viewer	The *"From"*-component can view something from the *"To"*-component.	A dropdown-list can *view the value* of a variable as its content, its selected index or its selected value.

All possible links



“From”-component qualifiers

	Invoker	Reactor	Provider	Viewer
	available, unavailable			
	received			
	loaded, unloaded			
				present (0 or 1)
	clicked		browse, camera	enabled (0 or 1)
	checked-changed		checked-value	checked (0 or 1)
	selected		selected-value	content, selected-index, selected-value
			image	image, mask
				text, present (0 or 1), font-color
	released		value	value
			selected-value	selected-value, present (0 or 1)
				text, present (0 or 1), font-color
				selected-index, enabled (0 or 1)
				text, present (0 or 1), font-color
			text	text, enabled (0 or 1)
	opened, closed			
		announce	data	
		announce	data	value
		show		enabled (0 or 1)
		toggle-play, play, stop		enabled (0 or 1)
		post		enabled (0 or 1)
	positive, negative	show, dismiss	input-text	enabled (0 or 1)

What will happen to the “To”-components

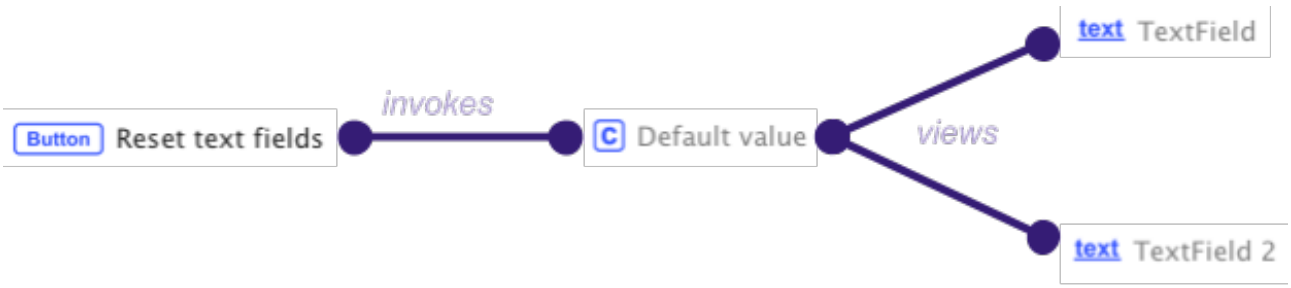
	Invoker	Reactor	Provider	Viewer
	Will be sent			
			Will get its data from the “From”-component	
		Will make the “From”-component react when received		
		Will make the “From”-component react when received		Will give its data to the “From”-component
	Will show			
	Will send its data to its viewers			Will give its data to the “From”-component
	Will send its data to its viewers		Will get its data from the “From”-component	Will give its data to the “From”-component
	Will show			
	Will play			
	Will show			

Using facts

There are two kinds of facts in PML: constants and variables. Both can *provide* their value to other components, and both can be invoked. When invoked, they will send their value to the components they provide to. Variables can also be viewers, i.e. they can get their value from other components. A property of variables can make them auto-invoke, meaning that they will invoke their new value to the components they provide to automatically when they get a new value.

Setting default values

A typical example when using constants is appropriate is when something has a *default value*. A parameter to a device might have a default value, that should be sent if it hasn't been updated from another component. Then we can have a constant provide that value to it, and let the application invoke the constant when loaded to give it the value when starting the application. Another example is if we want to empty a text field. Then we can have a constant with no value (or technically an empty string as its value) provide to the text field, and then have anything that should reset the text field invoke the constant.



A constant can be used to give text fields a default value when invoked, e.g. when a linked button is pressed

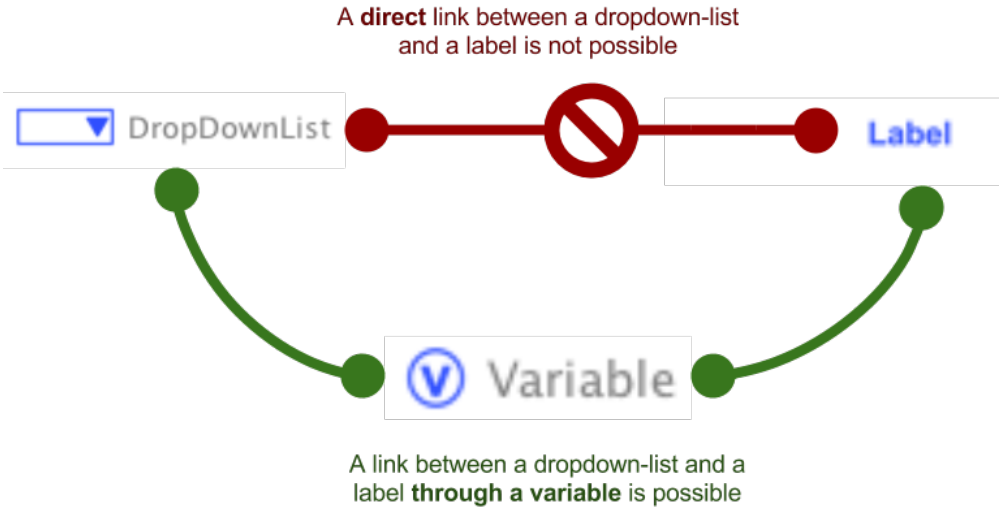
Enabling/disabling and hiding/showing components

A lot of components can view with the *enabled qualifier* (see the *"From"-component qualifiers*-diagram in the List of Links section to see which ones). By linking a constant (or a variable) with the "enabled" qualifier to any of these components, we can invoke the constant to enable or disable that component. Typically, we can link one constant with the value 0 to disable the component, and one with the value 1 to enable it.

In the same way, some components can be shown or hidden with the *present qualifier.* Since areas can be shown or hidden with the "present" qualifier, we can show or hide just about any component by just putting it in an area and showing or hiding the area. Similarly to with the "enabled" qualifier, we link constants (or variables) with the value 0 to hide the area or 1 to show it, and invoke the constant to change the visibility of the area.

Linking components that can't be connected directly

Variables can be convenient when connecting components that can't usually be connected. If we want the chosen value of a dropdown-list show in a label for example, it might not seem possible at first sight since we can't directly connect these components with a link (the label will get a red border when dragging the dropdown-list over it). We can however let the dropdown-list provide to a variable, and then let the label view the variable (make sure the variable is set to auto-invoke if you want the value to get updated automatically).



Layout

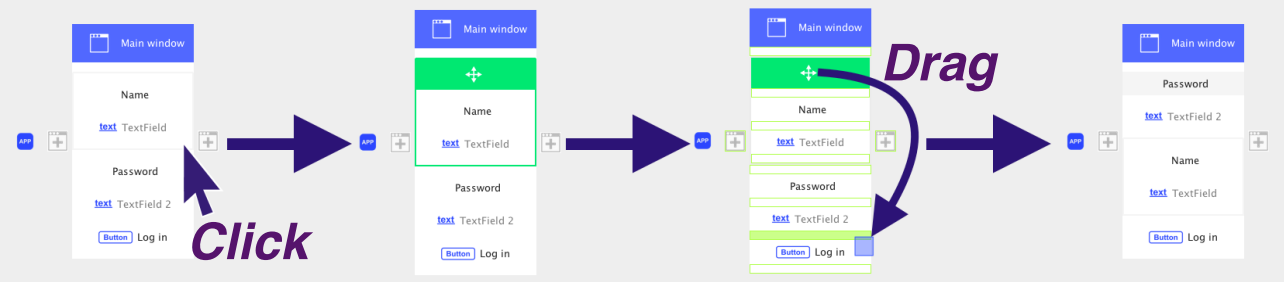
Moving components

Moving components is done by dragging components to an empty slot on the canvas, similarly to how we create components. Only components in the canvas can be moved however, i.e. not network units or facts/notifications.

Sometimes the component we move also have the option of linking to another component in the canvas. When moving such a component, we will get a choice of whether to move the component or to link it to a new component.

In order to move entire areas, we first need to select them by clicking in them, between their components. After having selected the area, an arrows-icon will become visible:

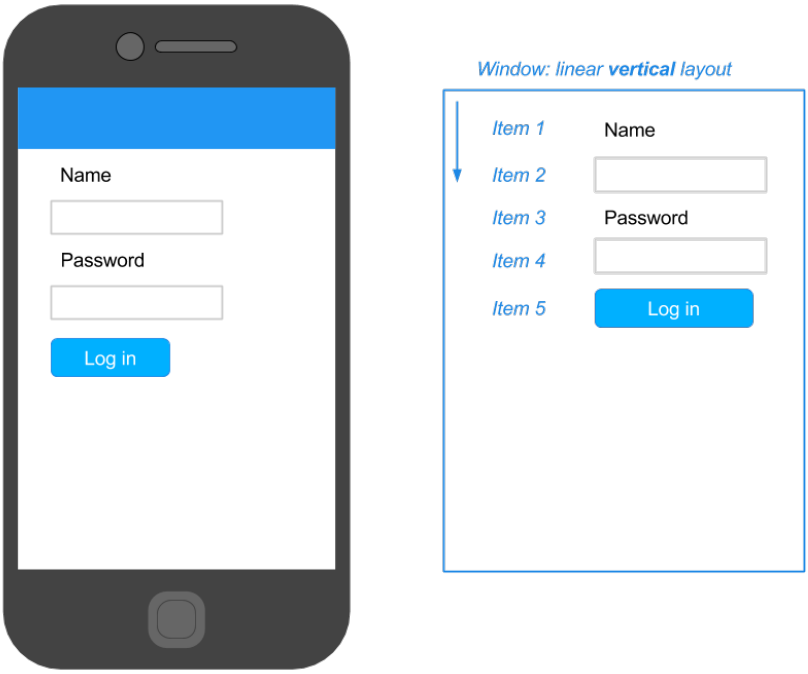
We can drag the arrows-icon like we would any other component to move the area (or create links to the area):



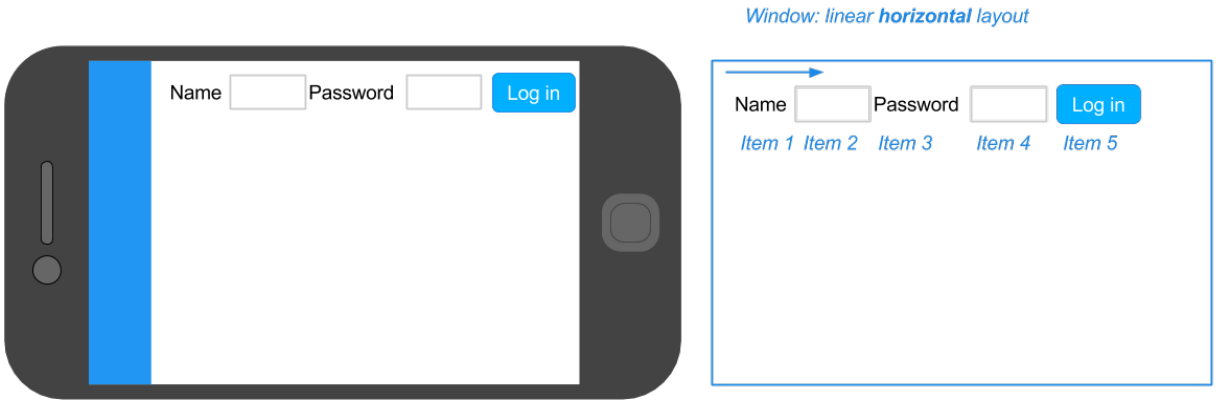
Layout types

PML has support for two major layouts, linear layout and grid layout. In a linear layout, the components are placed one after the other, either horizontally or vertically. In a grid layout, the components are laid out in a grid. The layout can be changed for windows, areas and radio-button groups. The layout can be edited along with the other properties in the properties dialog (which can be viewed by selecting the component and clicking the “Properties”-button).

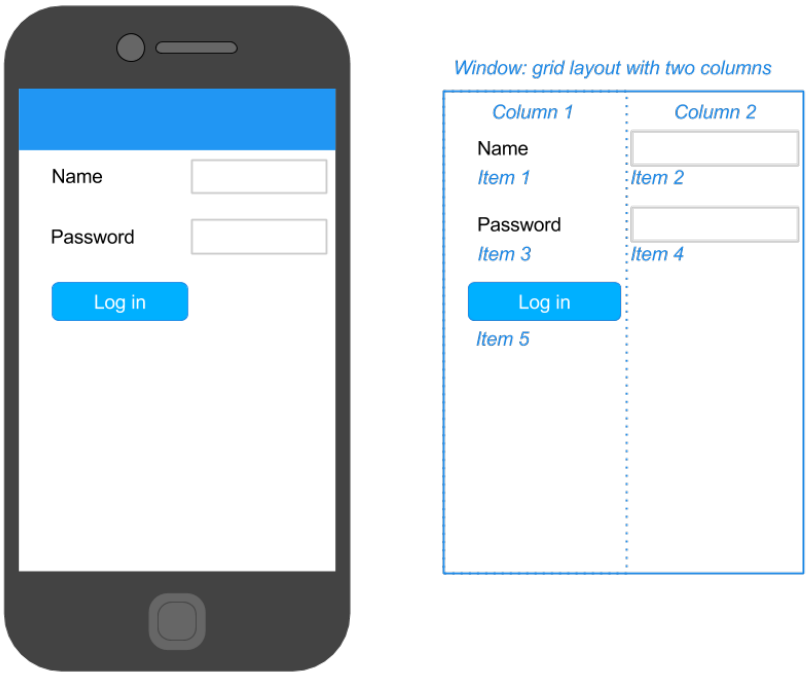
Below is an example of a window with a linear *vertical* layout. The components of the window are laid out one after the other vertically. To the left is what the finished application will look like, and to the right is a sketch of the layout of the components:



Below is an example of a window with linear *horizontal* layout. The components are laid out one after the other horizontally:



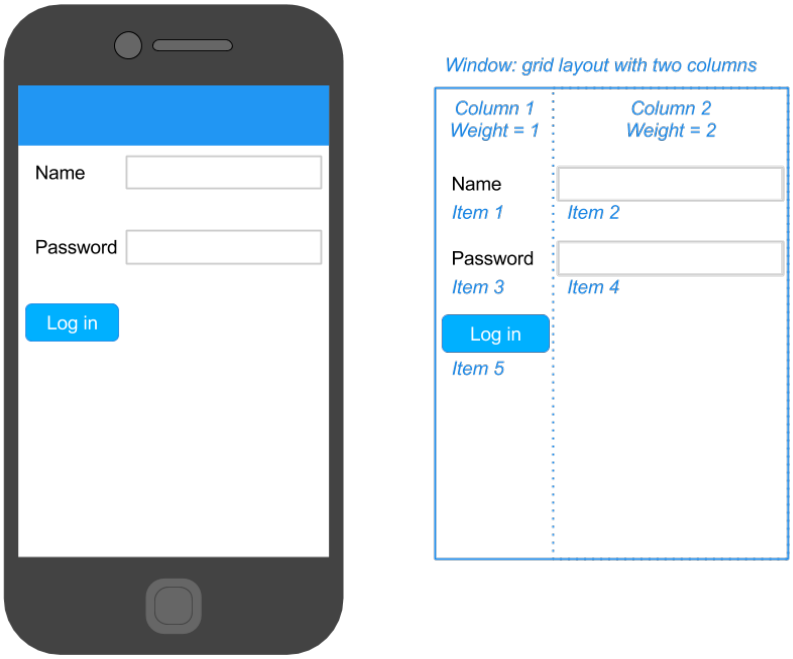
Below is an example of a grid layout with two columns. The components are laid out left to right, but will begin from the left again (one step down) after having reached the number of columns specified (so if the number of columns is two, items number 3 and 5 will begin again from the left).



For further control of the grid layout, we can specify *column weights* for the columns. The column weights lets us choose how wide each column should be compared to the rest. For example, a column with weight 2 will be twice as wide as a column with weight 1. Note that column weights are not reflected in the PML Editor canvas, but will show when running the application.

Generally, for every column, the *relative width of a column will be the weight of that column divided by the total weight*. If we for example have three columns with the weights 1, 3 and 6 (which means the total weight is 1 + 3 + 6 = 10), the first column will be 1/10 wide, the second 3/10 and the third 6/10. By default, we all columns have the weight 1, which means they will have equal width.

The picture below shows a grid layout with two columns with the weights 1 and 2 (and the second column is thus twice as wide as the first):



For the mathematically inclined, column weights can be explained by:

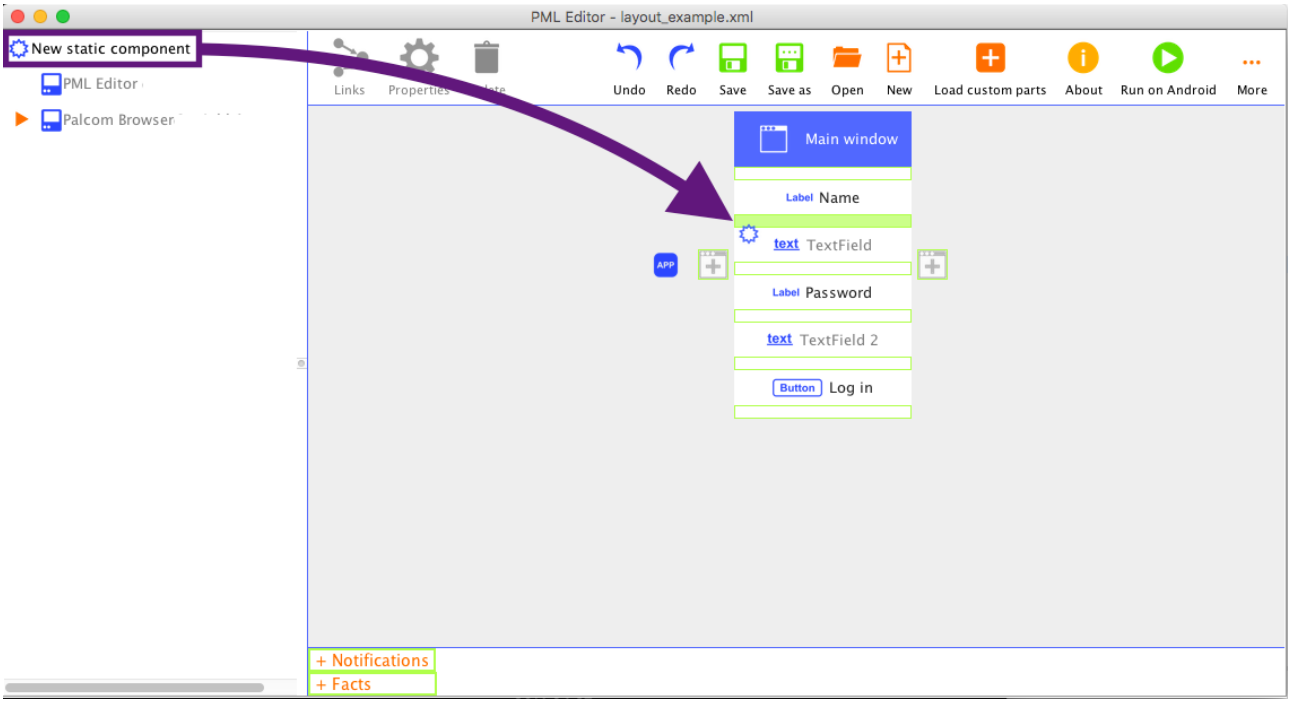
$$\textit{for } n \textit{ columns } c_1, c_2 \ldots c_n \textit{ with weights } w_1, w_2 \ldots w_n,$$
$$\textit{the width of a column } c_x,\ 1 \leq x \leq n, \textit{ is } \frac{w_x}{w_1 + w_2 + \ldots + w_n}$$

The layout of windows, areas and radio-button groups can be set by selecting the component, and clicking the “Properties”-button. For both linear and grid layout, we can set also set the

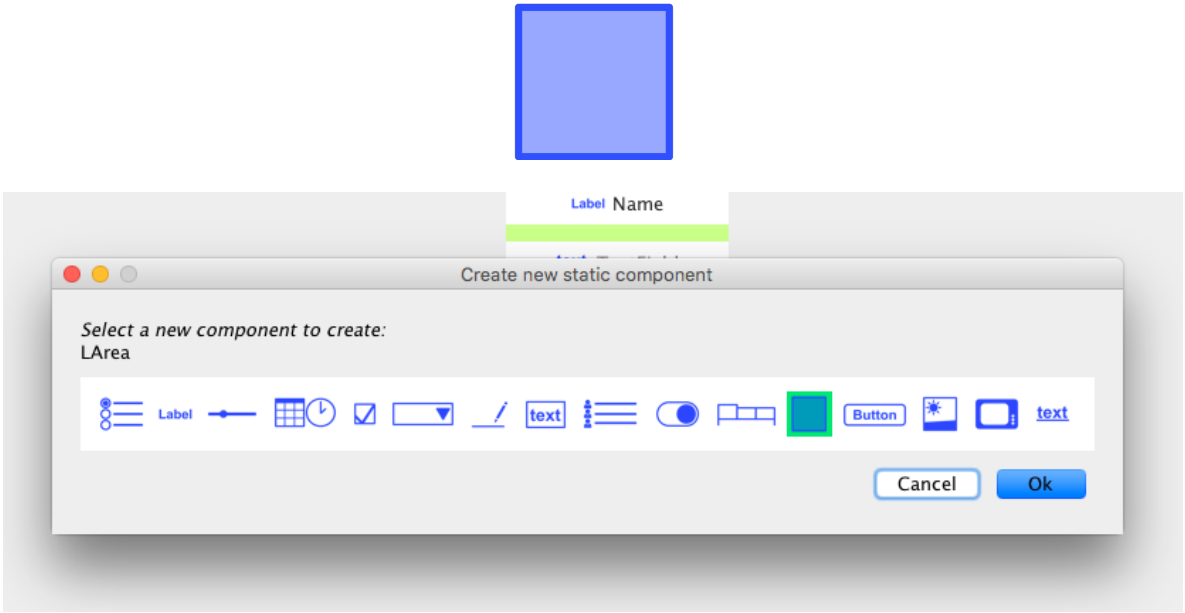
Areas and layout nesting

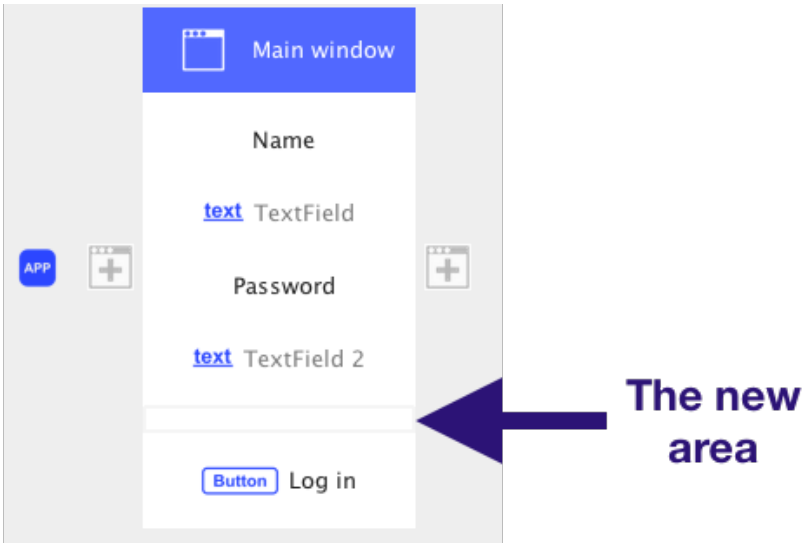
The *Area* component is a *container component*, i.e. a component that can contain other components. Areas provide a way to structure our application, and to nest different layouts within each other.

Areas are created like other static components, by dragging the “New static component”-label to where you want the area in the canvas:



And selecting “Area” in the dialog that pops up:



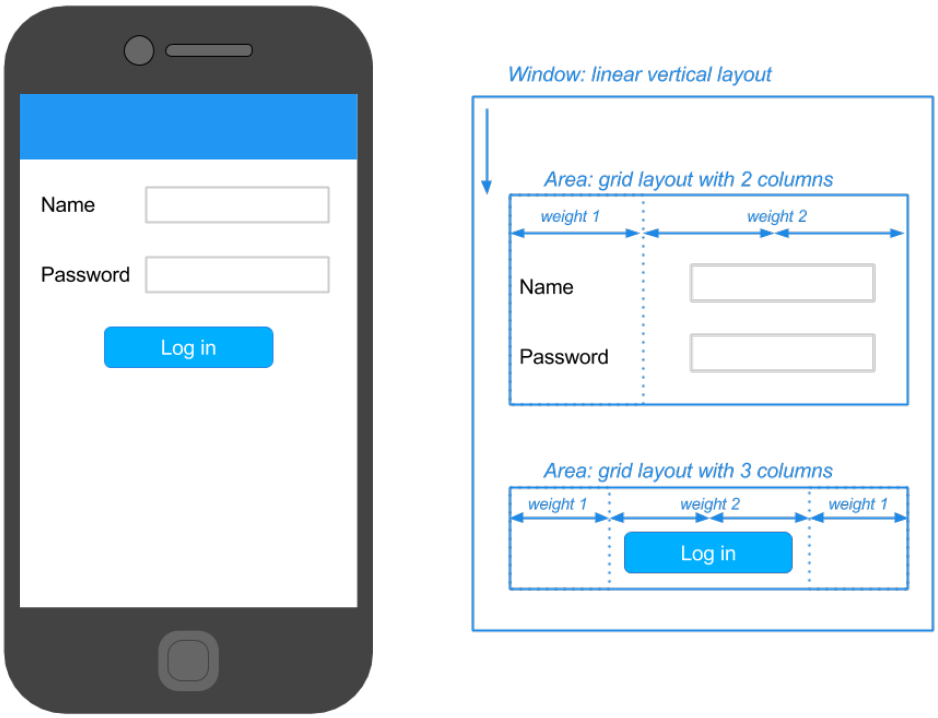


We can then move new components into the new area by dragging them.

Areas enable us to nestle layouts within each other, to create more complex layouts. In the example below, we have three different layouts:

1. The enclosing window has a linear vertical layout
2. The first area has a grid layout with two columns of weight 1 and 2
3. The second area has a grid layout with three columns of weight 1, 2 and 1.

Note that the two columns beside the “Log in”-buttons have *empty labels* (labels without text) in them used for spacing.



Spacing with empty labels

Empty labels are sometimes handy when positioning components in a grid layout, to allow for spacing. We can for example align a component to the right in an area by using a grid layout with two columns, and putting an empty label in the first. An empty label is a normal label, but with no text, which makes it invisible while still filling a column in a grid layout. To create an empty label, we can drag the “New static component”-label to where we want the empty label, and select “Label” in the dialog that pops up (like when creating any static component). Then just leave the “Text”-field blank in the properties dialog.

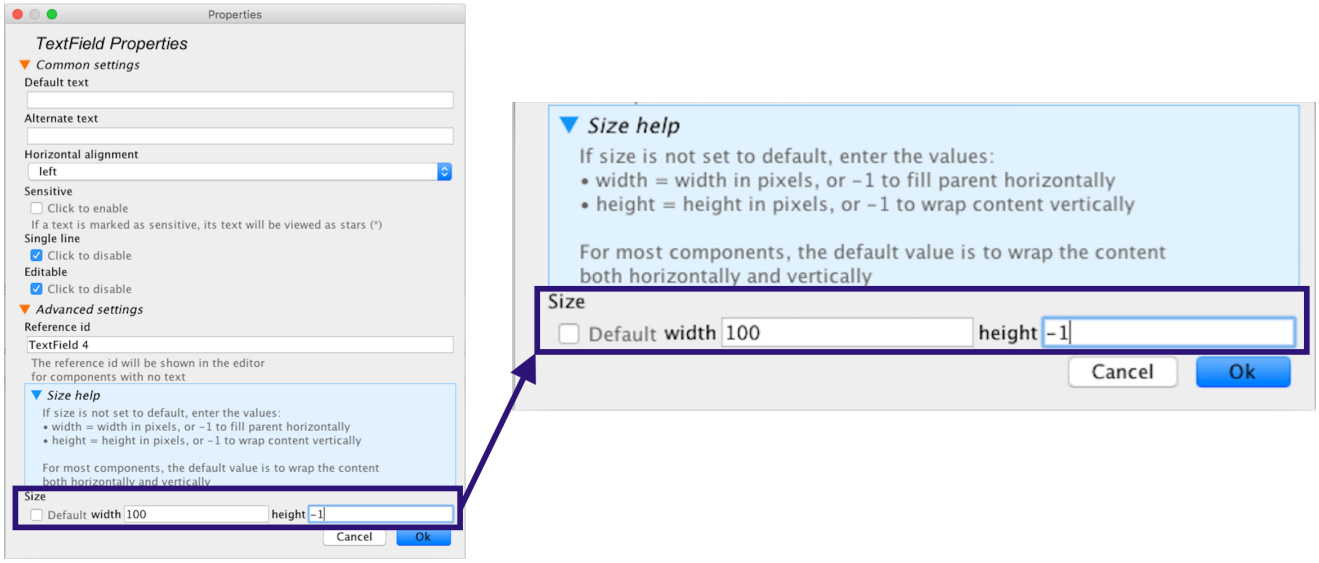
Setting the size of components

An additional setting that comes in handy when structuring the application is the *size* property. All components in the canvas have a size property, which can be found in the bottom part of the properties dialog for the component under the *Advanced settings* title. By unselecting the *default* checkbox, custom input is enabled. The first field is for width and the second for height.

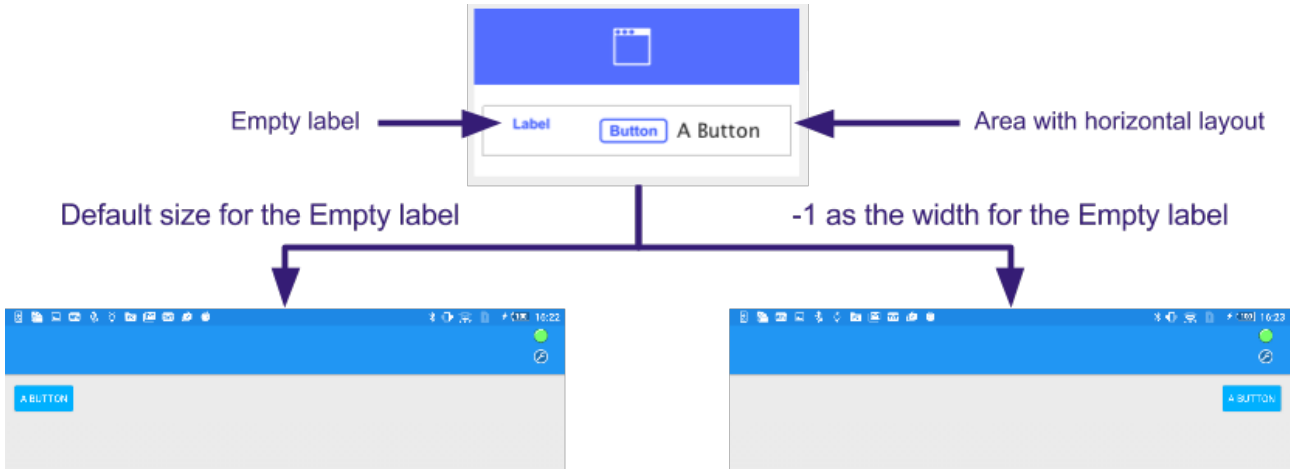
By default, most components will take up as much space as needed vertically and horizontally. Areas are an exception to this: they will fill their parent component horizontally.

The width and height can be set to an explicit value in pixels. In addition to this, the width and height can be set to "-1". Setting the *width* to "-1" will make the component *fill its parent horizontally* (or, if other components have this setting in the container: share any leftover space with the other components). Setting the *height* to "-1" will make the component *take up as much space as is needed vertically*. Note that the size will not be reflected in the PML Editor canvas, but will take effect when running the application.

Below, we have set the size of a text field to be 100 pixels wide horizontally and to take up as much space as needed vertically:



Another example of using the "-1" in the size property:

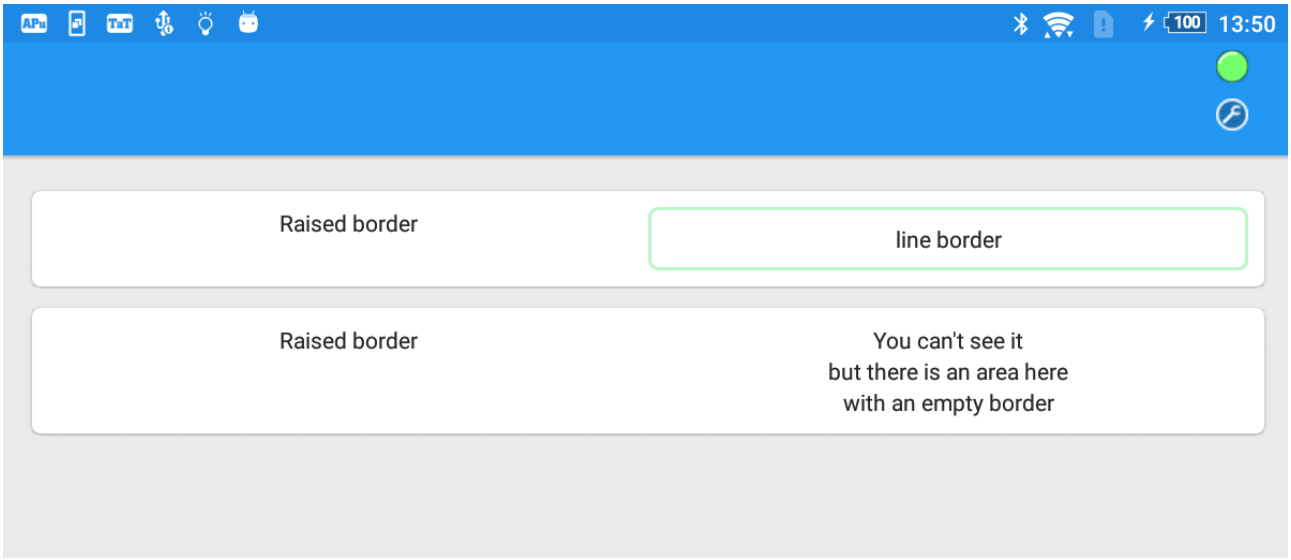


Borders on Android

Areas can have borders, to make them stand out more from the surrounding application. There are three kinds of borders compatible with the Android back-end for PML:

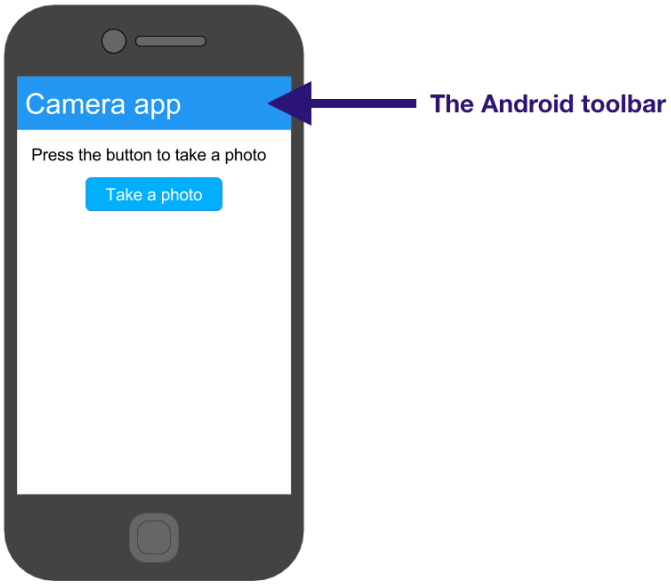
- *Raised* - Raised borders make the area get a white background as opposed to the usual grey of the application, and the area will appear to be raised compared to the background. This border is typically used for the main areas of each window.
- *Empty* - Empty borders are invisible. Empty borders are used for areas that mainly serve as a layout tool. This is the default border.
- *Line* - An area with a line border will get a green line surrounding it, and the background will be white.

In addition to these three, PML also allows for the *lowered* border, but in the Android back-end, this only serves as a way to create custom Android toolbars (see the *Creating a custom Android toolbar* section for more info).

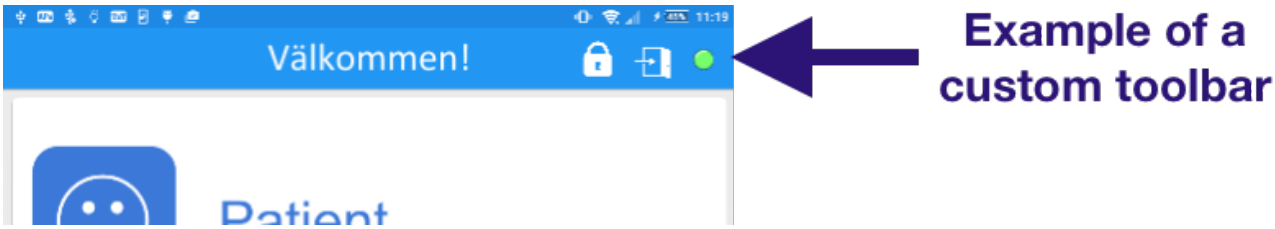


Creating a custom Android toolbar

The Android toolbar is the blue stripe in the top of every window, which displays the window title per default. The title can be edited in the window settings, but there is also a way to create a fully custom toolbar, containing any components you want.



To create a custom toolbar for a window, make sure that *the window is using a linear layout* (if you want the contents of the window to be in a grid layout, use an area within the window). Next, create an *area* as the *first component* in your window (at the top of the window for linear *vertical* layout, or at the far left for linear *horizontal* layout). Set the border of that area to be *lowered*. The area will then be shown as a toolbar, instead of the standard one. You can place any components in the area (and use any layout), but it's recommended that you try to keep the toolbar from becoming bigger than the default toolbar.



Running PML files

Introduction

The PML Editor loads and saves xml-files, which can also be run on an Android device. When you save an application you are making in the PML Editor, you will thus get a file that can be both loaded back into the PML Editor (for further editing) and run on an Android device.

The file from the PML Editor needs however to be run *inside of an application on the Android device*. The name of this application is AndroidPUIDI, and it looks like this:

To run your applications on an Android device, you thus first need to install AndroidPUIDI. After having done that, the easiest way to run your application is to connect your Android device to your computer (with a USB-cable) and press the “Run on Android”-button.

Currently, AndroidPUIDI is the only available back-end for PML files. More back-ends are however planned to be developed in the future.

Installing AndroidPUIDI

This section will give a brief introduction on how to install and use the AndroidPUIDI (“*APu*”) app. To make full use of the potential of the app, the *TheAndroidThing* (TaT) app should be installed on the same device.

Installing TheAndroidThing is like installing any other Android application package file (APK). This can be done in several ways. Below follows one way to do it. Other, possibly simpler ways (depending on the conditions) can be found by searching the web.

1. Modify the Android device’s settings to enable installation from sources other than Google Play (former Android Market). Under “Settings”, select “Applications” and then check the box next to “Unknown sources”.

2. On the Android device, download the APK file from this link: https://goo.gl/FRZ0nH.

3. Install the app by clicking on the downloaded file in any file explorer (e.g. ES File Explorer: goo.gl/RPcGB4) on the Android device.

Connectivity

By default, the Palcom device attached to APu upon startup is only connected to an IPC MAO, i.e. a media abstraction object that can communicate with other device on the same Android device. This should be enough for APu and the device attached to TaT to discover each other. Then, by enabling routing in TaT, and adding more interfaces (MAO) to its device, the reach of APu can be extended outside the Android device.

If TaT is not installed on the device, the reach of APu can also be extended by opening the preferences menu. Push the three vertical dots (⋮), then "Preferences", and then check the box next to "IPv4 pseudo interface". The device attached to APu should now be discoverable from any other device connected to the same IP network (e.g. WiFi).

Usage Overview

- Start AndroidPUIDI by tapping on the icon labeled “PUIDI":

- Depending on the number of installed descriptions (see section “Installing Descriptions”) one of three things will happen:
  - *No descriptions*: a warning, followed by the option to manually search for a description file to load.
  - *One description*: the description is automatically loaded.
  - *Multiple descriptions*: A list with the names of all installed descriptions is presented, prompting the user to select which one to load.

- Tab on the menu button (three vertical dots [⋮]) then “Preferences”, to bring up the preference menu.

Running PML files from the editor

Running PML files from the PML Editor is simple. Connect an Android device with the AndroidPUIDI application to your computer with a USB-cable, and click the “Run on Android”-button:

The file will be installed and run automatically. Note that your file will be saved. If you are using any custom parts, these will need to be installed beforehand.

The first time you click the “Run on Android”-button, you will need to locate the program “adb” (*Android Debug Bridge*) on your computer. If you don’t have adb installed, it can easily be found and downloaded free of charge on the Internet.

If, after having chosen a location for adb, need to change the location, you can press shift while clicking the “Run on Android”-button, or pressing the keys *control + shift + R* on Windows/Linux or *command + shift + R* on Mac.

Installing descriptions from a PalCom Browser

To install a description on AndroidPUIDI (APu), start by locating its XML-file on the local machine (i.e. the file you saved from the PML Editor). Open APu; ignore (back button [↩]) any initial prompts. The description manager service should already be enabled. Make sure the device of APu is visible from the development computer (see section “Connectivity”). On the computer, open the PalcomBrowser tool. Locate the PalCom device corresponding to APu, and use the “Description Manager” service to install the description XML-file. Please refer to the user’s manual for the PalcomBrowser tool on how to do this.

Custom parts

Creating custom parts

New PML parts can be created by third parties and added dynamically to the PML Editor and AndroidPUIDI (APu) for use in PML applications. To start creating custom PML parts, include the *JavaFrontEndPUIDI* library (pml-ife.jar) into your Java project (http://fileadmin.cs.lth.se/cs/personal/bjorn_a_johnsson/PML/jfe-pdi.jar). Create a new class that extends either *LPart* or *LContainerPart* depending on whether the new part should be able to contain other parts or not. Unfortunately, custom *container* parts are not supported by the PML Editor at this time.

Override the methods build(AbstractBuilder):Object and toXmlElement(Document):Element. The Javadoc explains their function. The new class must implement the interface *LCustom*. Again, refer to the Javadoc for clarification when implementing.

Example implementations of custom parts that can be used for inspiration, guidance or as starting points for new parts can be found in the "CustomAndroidParts" project (http://fileadmin.cs.lth.se/cs/personal/bjorn_a_johnsson/PML/CustomAndroidParts.zip). This project includes both the source code for five example parts that showcase different ways to implement custom parts, and PML descriptions that can be used to test the new parts.

Note that if you want a *custom icon* for your custom part when using it in the PML editor, you may include a file called *[custom part class name].png* (e.g. *ButtonExamplePart.png*) in your custom part jar file. If this file isn't present, a standard icon is used instead:

It's recommended that the icon is approximately 28 x 28 pixels (or less), and that the colors used are *#304ffe* (blue), *#304ffe7c* (semi-transparent version of the blue color) and transparency.

Loading custom parts into the PML Editor

You can load custom parts into the editor by clicking the “Load custom parts”-button, and locating the custom part jar file:

Once loaded, you can use the custom parts like any of the other components. To see what custom parts are loaded, click the “About”-button:

Installing custom parts on AndroidPUIDI

To install a custom part on AndroidPUIDI (APu), see the section “Installing descriptions from a PalCom Browser”. Instead of using the “Description Manager”, use the “Custom Part Manager” service. Once installed, the new part can be used in any PML description by setting the class attribute of a part element to the canonical name of the part's class, e.g.:

```
<part class="se.lth.cs.palcom.bjornaj.custom.ButtonExamplePart"/>
```

Troubleshooting

Network devices don’t appear

Note that devices need to be turned on and connected to the same network the computer is connected to, in order to be discovered by the editor. Also note that if more than one instance of the editor is run at the same time, it will be confused and the network devices will be split arbitrarily between the instances.

I can’t link components that I want to be able to link

Not all components can be linked. There are however some tricks that can be used for linking components that can’t be linked directly. Check out the *Using facts*-section for more information. In some cases however, you might need to create custom parts if you want components with specific behavior.