qooxdoo Documentation

Release 1.3.1

qooxdoo developers

April 06, 2011

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 About

qooxdoo (pronounced ['kuksdu:]) is a comprehensive and innovative framework for creating desktop-style web applications, often called rich internet applications (RIAs). Leveraging object-oriented JavaScript allows developers to build impressive cross-browser applications. No HTML, CSS nor DOM knowledge is needed. qooxdoo includes a platform-independent development tool chain, a state-of-the-art GUI toolkit and an advanced client-server communication layer. It is Open Source under an LGPL/EPL dual *license*.

1.2 Framework

qooxdoo is entirely class-based and tries to leverage the features of object-oriented JavaScript. It is fully based on namespaces and does not extend native JavaScript types to allow for easy integration with other libraries and existing user code. Most *modern browsers* are supported (e.g. Firefox, Internet Explorer, Opera, WebKit/Safari) and it is free of memory leaks. It comes with a comprehensive API reference, that is auto-generated from Javadoc-like comments. The fast and complete JavaScript parser not only allows doc generation, but is an integral part of the automatic build process that makes optimizing, compressing, linking and deployment of custom applications very user-friendly. Internationalization and localization of applications for various countries and languages is a core feature and easy to use. more ...

1.3 GUI Toolkit

Despite being a pure JavaScript framework, qooxdoo is quite on par with GUI toolkits like Qt or SWT when it comes to advanced yet easy to implement user interfaces. It offers a full-blown set of widgets that are hardly distinguishable from elements of native desktop applications. Full built-in support for keyboard navigation, focus and tab handling and drag & drop is provided. Dimensions can be specified as static, auto-sizing, stretching, percentage, weighted flex or min/max or even as combinations of those. All widgets are based on powerful and flexible layout managers which are a key to many of the advanced layout capabilities. Interface description is done programmatically in JavaScript for maximum performance.

No HTML has to be used and augmented to define the interface. The qooxdoo developer does not even have to know CSS to style the interface. Clean and easy-to-configure themes for appearance, colors, borders, fonts and icons allow for a full-fledged styling.

1.4 AJAX

While being a client-side and server-agnostic solution, the qooxdoo project includes different communication facilities, and supports low-level XHR requests as well as an RPC API. An abstract transport layer supports queues, timeouts and implementations via XMLHttpRequest, Iframes and Scripts. Like the rest of qooxdoo it fully supports event-based programming which greatly simplifies asynchronous communication.

1.5 More Information (online)

- FAQ
- License
- Framework Features
- Release Notes
- Roadmap
- Developers
- Committers Guide
- Media Download

CHAPTER

TWO

GETTING STARTED

2.1 Requirements

Here are the requirements for developing and deploying a qooxdoo application. A typical qooxdoo application is a JavaScript-based "fat-client" that runs in a web browser. It does not enforce any specific backend components, any HTTP-aware server should be fine. The framework comes with a powerful tool chain, that helps both in developing and deploying applications.

It is very straightforward to satisfy the requirements for those three topics (client, server, tools).

2.1.1 Client

A qooxdoo application runs in all major web browsers - with identical look & feel:

	Internet Explorer 6+
	Firefox 2+
	Opera 9+
	Safari 3+
	Chrome 2+

Not only the *end users* of your application benefit from this true cross-browser solution. As a developer you can also pick *your* preferred development platform, i.e. combination of browser and operating system. Most built-in developer *Tools* (e.g. for debugging, profiling) work cross-browser as well.

2.1.2 Server

Developing a qooxdoo application does not require a server. Its static application contents (initial html file, JavaScript files, images, etc.) may just be loaded from your local file system.

Of course, for the actual deployment of your final app you would use a web server to deliver the (static) contents. For developing a qooxdoo app it is not a prerequisite to setup a web server, so you can start right away on your local computer.

Any practical qooxdoo client application will communicate with a server, for instance to retrieve and store certain application data, to do credit card validation and so on. qooxdoo includes an advanced *RPC mechanism* for direct calls to server-side methods. It allows you to write true client/server applications without having to worry about the communication details. qooxdoo offers such *optional* RPC backends for Java, PHP, Perl and Python. If you are missing your favorite backend language, you can even create your own RPC server by following a generic *server writer guide*.

If you already have an existing backend that serves HTTP (or HTTPS) requests and you do not want to use those optional RPC implementations, that's fine. It should be easy to integrate your qooxdoo app with your existing backend using traditional AJAX calls.

2.1.3 Tools

qooxdoo comes with a platform-independent and user-friendly tool chain. It is required for *creating and developing* a qooxdoo application. It is *not* needed for running an application.

The tool chain only requires to have Python installed. Use a standard **Python 2.x** release, version 2.5 or above. **Python 3** is currently not supported! As a qooxdoo user you do not need any Python knowledge, it is merely a technology used internally for the tools. Python comes either pre-installed on many systems or it can very easily be installed:

Windows

It is trivial! Just download and install the excellent **ActivePython** package. Its default settings of the installation wizard are fine, there is nothing to configure. (It is no longer recommended to use the Windows package from Python.org, as this requires additional manual *configuration*).

Cygwin

Cygwin can be used as an optional free and powerful Unix-like environment for Windows. You won't need a native Python installation, just make sure to include Cygwin's **built-in** Python as an additional package when using Cygwin's setup program.

Mac

Python is **pre-installed** on Max OS X. No additional software needs to be installed, but on older systems it might need an update.

Linux

Python often comes **pre-installed** with your favorite distribution. If not, simply use your package manager to install Python.

2.2 Hello World

This tutorial is a step-by-step instruction on how to get started with qooxdoo by creating your very first application.

2.2.1 Setup the Framework

Requirements

Please make sure to have read the detailed *Requirements*. To recap, there are only a few requirements for full-featured qooxdoo application development:

- *client*: any major web browser
- *server*: any HTTP-aware backend. During development the local file system should also be ok (*)
- *operating system*: any
- *tools*: Python required

(*) Developers using Chrome should note that there is a known issue loading reasonably complex qooxdoo applications (such as the API viewer or the demo browser) via the file:// protocol. It is recommended to Chrome users to use the HTTP protocol, even while developing.

Download

Go to the Download section and grab the latest stable Software Development Kit (SDK).

Installation

Unzip the SDK archive.

2.2.2 Create your Application

It is easy to setup your own application using the platform-independent script `create-application.py`. It will create a skeleton application in a directory you specify, that is automatically configured to work with your version of the qooxdoo framework.

To create a new skeleton with `create-application.py` you will need to follow some initial *platform-dependent* steps - even when the rest of your development is independent of the platform. Please see the appropriate section below for *Windows* , *Cygwin* or *Mac* , *Linux*

Note: If you have any problems setting up the qooxdoo tool chain, please see some additional help for *troubleshooting*.

Windows

Installing ActivePython for Windows is trivial. Now let's create an application named `custom` in `C:`, with the qooxdoo SDK available at `C:\qooxdoo-1.3.1-sdk`:

```
C:\qooxdoo-1.3.1-sdk\tool\bin\create-application.py --name=custom --out=C:
```

Cygwin

To create your application `custom` to `C:`, with the qooxdoo SDK available at `C:\qooxdoo-1.3.1-sdk`, call the script as follows:

```
/cygdrive/c/qooxdoo-1.3.1-sdk/tool/bin/create-application.py --name=custom --out=C:
```

Mac , Linux

To create an application `custom` in your home directory, change to your home directory (just `cd`). With a qooxdoo SDK available at `/opt/qooxdoo-1.3.1-sdk`, call the script as follows:

```
/opt/qooxdoo-1.3.1-sdk/tool/bin/create-application.py --name=custom --out=.
```

2.2.3 Run your Application

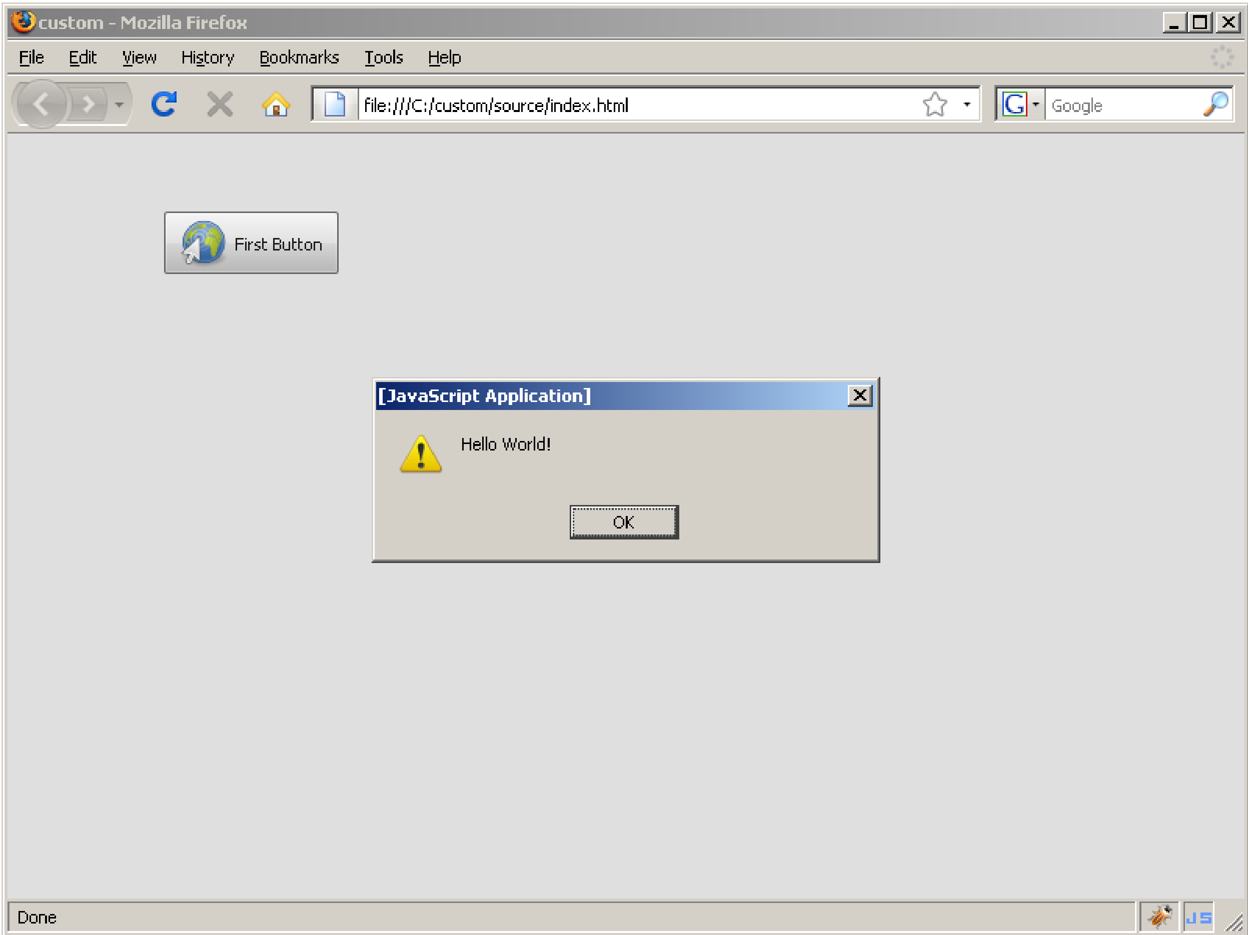
Now that your application is setup, lets generate a version that can be opened in your browser. Move to the newly created application directory and kick off the automatic build process:

```
cd C:/custom
generate.py source-all
```

Under non-Windows systems you might have to prefix the command with the local directory, i.e. execute `./generate.py source-all` instead.

Please note, that the additional `source-all` target was introduced with qooxdoo 0.8.1. The regular `source` target now only includes those qooxdoo *classes* that are actually required by your app, not all the source classes.

After the application has been generated, open `source/index.html` file in your web browser to run your application and click the button:



2.2.4 Write Application Code

The folder `source/class` contains all your application classes. When starting with a newly created application, there is only a single file `custom/Application.js`. Open it in your favorite editor or IDE.

The method `main()` contains the entire code of your little skeleton app. Even if you haven't done any qooxdoo programming before, you should be able to figure out what the code does. Get familiar with the code and change it, e.g. modify the label of the button, move the button to another position or add a second button.

To see the changes, you just have to refresh your document in the browser, e.g. by hitting `F5`. During development there usually is no need to re-generate this so-called "source" version of your app. Only if you later introduce new classes or if dependencies between classes change, you would have to regenerate your app. To do so, execute `generate.py source-all` (to include all source classes) or `generate.py source` (to only include the required classes) before refreshing your browser.

2.2.5 Debugging

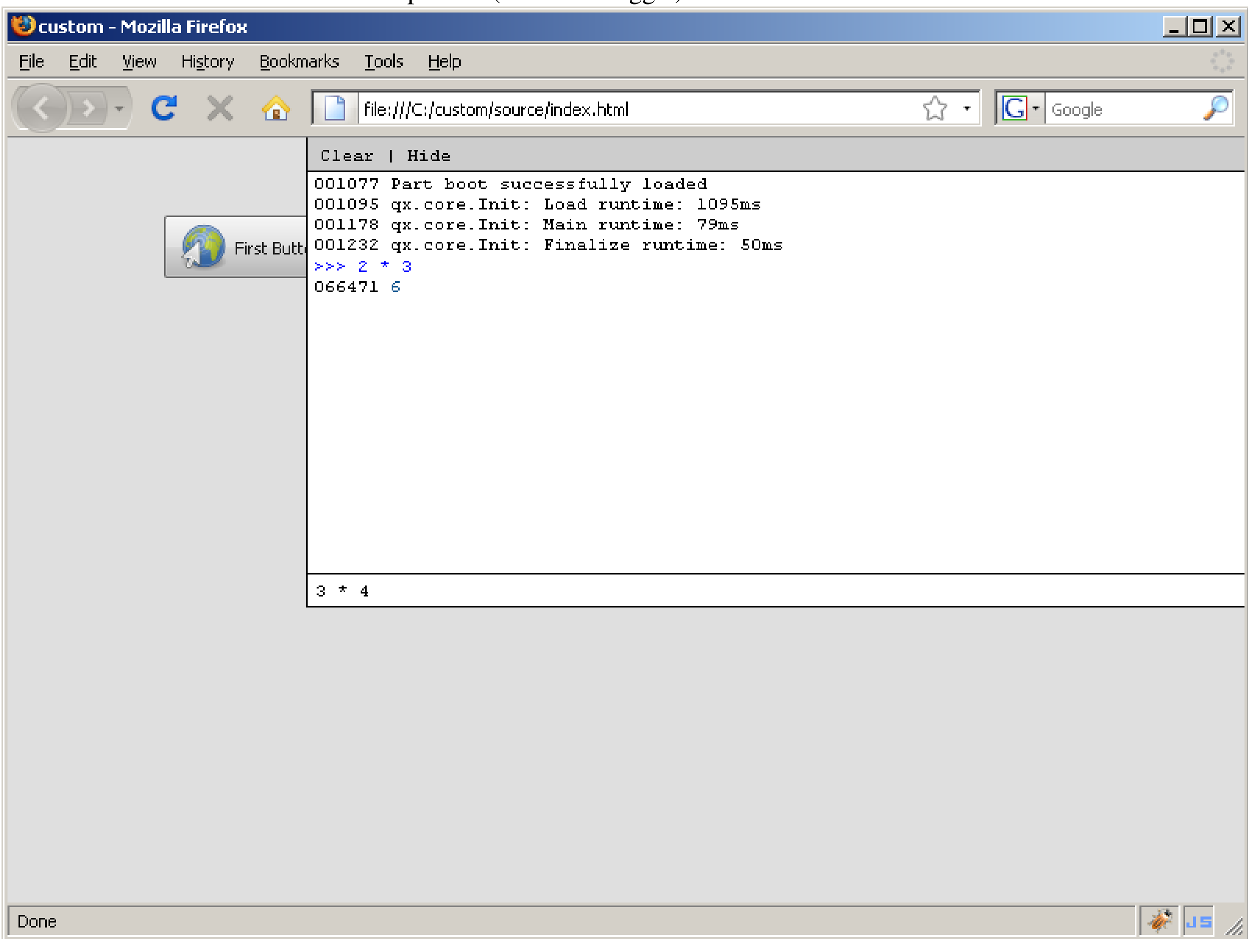
In your newly created application you have certainly noticed the following code:

```
if (qx.core.Variant.isSet("qx.debug", "on"))
{
  qx.log.appender.Native;
```

```
  qx.log.appender.Console;
}
```

This code turns on two different ways of "logging", i.e. capturing and printing out information about the operation of your application.

`qx.log.appender.Native` uses the native logging capabilities of your client if available, e.g. Firebug in Firefox (use `F12` to toggle). If your browser doesn't come with developer-friendly logging, `qx.log.appender.Console` provides such a feature for *all* browsers: the console prints out the log messages in an area inside your browser window. It also includes an interactive JavaScript shell (use `F7` to toggle):



The reason for enclosing the two logging classes in a so-called "debug" variant is explained in more detail in the next section. It ensures that logging is only turned on in the development version (i.e. "source" version) of your app. It will automatically be turned off in the final version of your app that is to be deployed:

2.2.6 Deployment

The development version of a qooxdoo app is called the "source" version, the deployment version of an app is called "build" version. It is easily generated by executing

```
generate.py build
```

After successful completion let the browser open `index.html` from the newly created `build` folder. Although you probably won't see a difference between this deployment version of your app and the previous "source" version, it should have started up faster.

Unlike the “source” version, with its numerous unmodified JavaScript files, the “build” version only has to load a single, optimized JavaScript file.

Manually creating such a “custom build” from your application class (or classes) would have been a very tedious and complex job. In fact most other JavaScript libraries do provide built-in support to automate this task. Building your app strips off unneeded whitespaces and comments, optimizes and reorganizes your code, uses a JS linker to only include classes that your application needs, and many more refinements and optimizations as well.

A lot of debugging code is also removed when a “build” is generated, that would only be useful during development of your application, e.g. printing out informative warnings or coding hints. Just like the logging code in the section above, you can put arbitrary code into such “variants”, which may then be automatically removed during “conditional compilation” of the build process. This lets you receive information on your app when you’re developing it, but removes this for your final code, so your end users don’t see it.

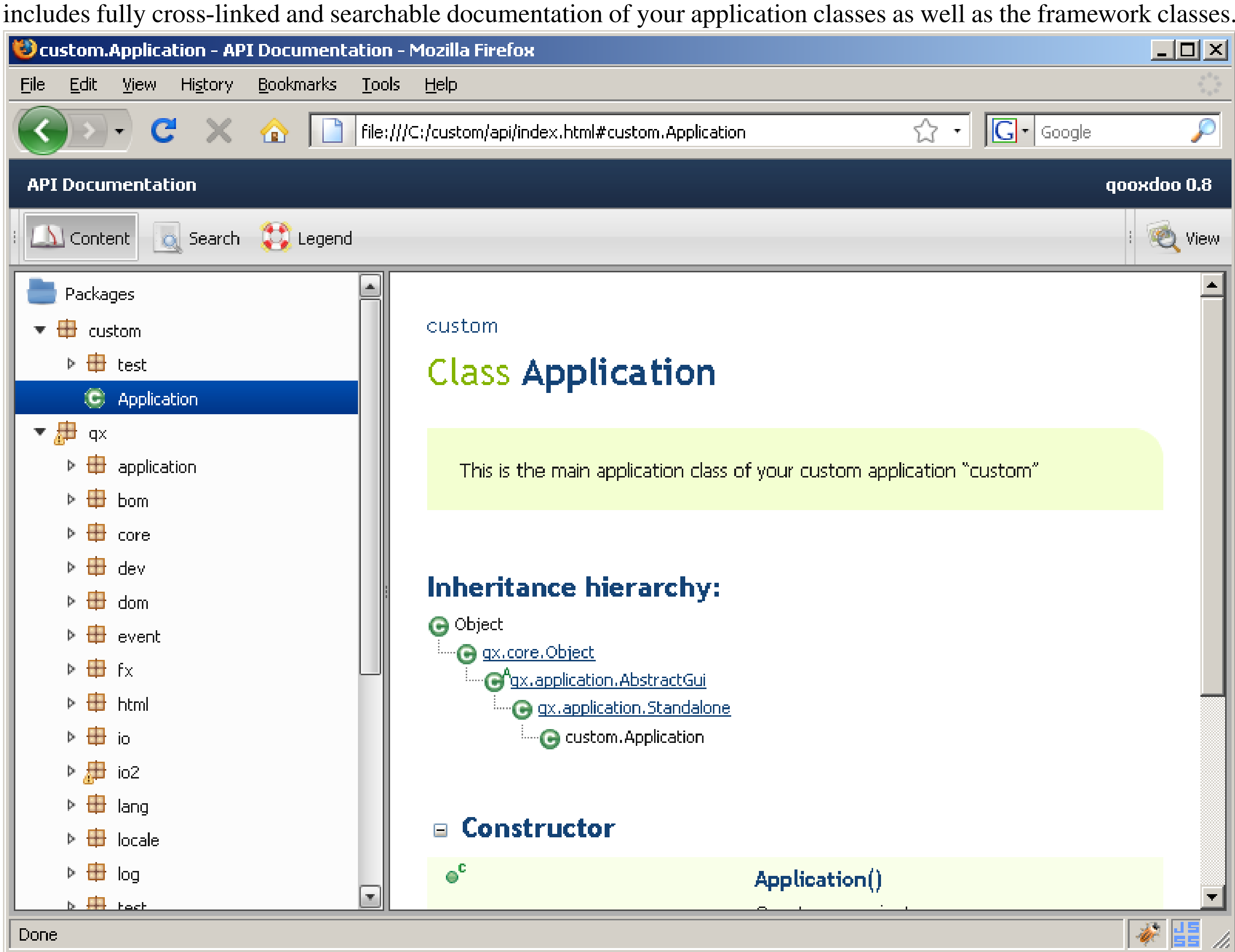
2.2.7 API Reference

qooxdoo supports inline comments that are similar to Javadoc or JSDoc comments. They allow for JavaScript and qooxdoo specific features, and look like `/** your comment */`.

From those comments a complete, interactive API reference can be generated:

```
generate.py api
```

To start the “API Viewer” application, open `index.html` from the newly created `api` folder in your browser. It includes fully cross-linked and searchable documentation of your application classes as well as the framework classes.

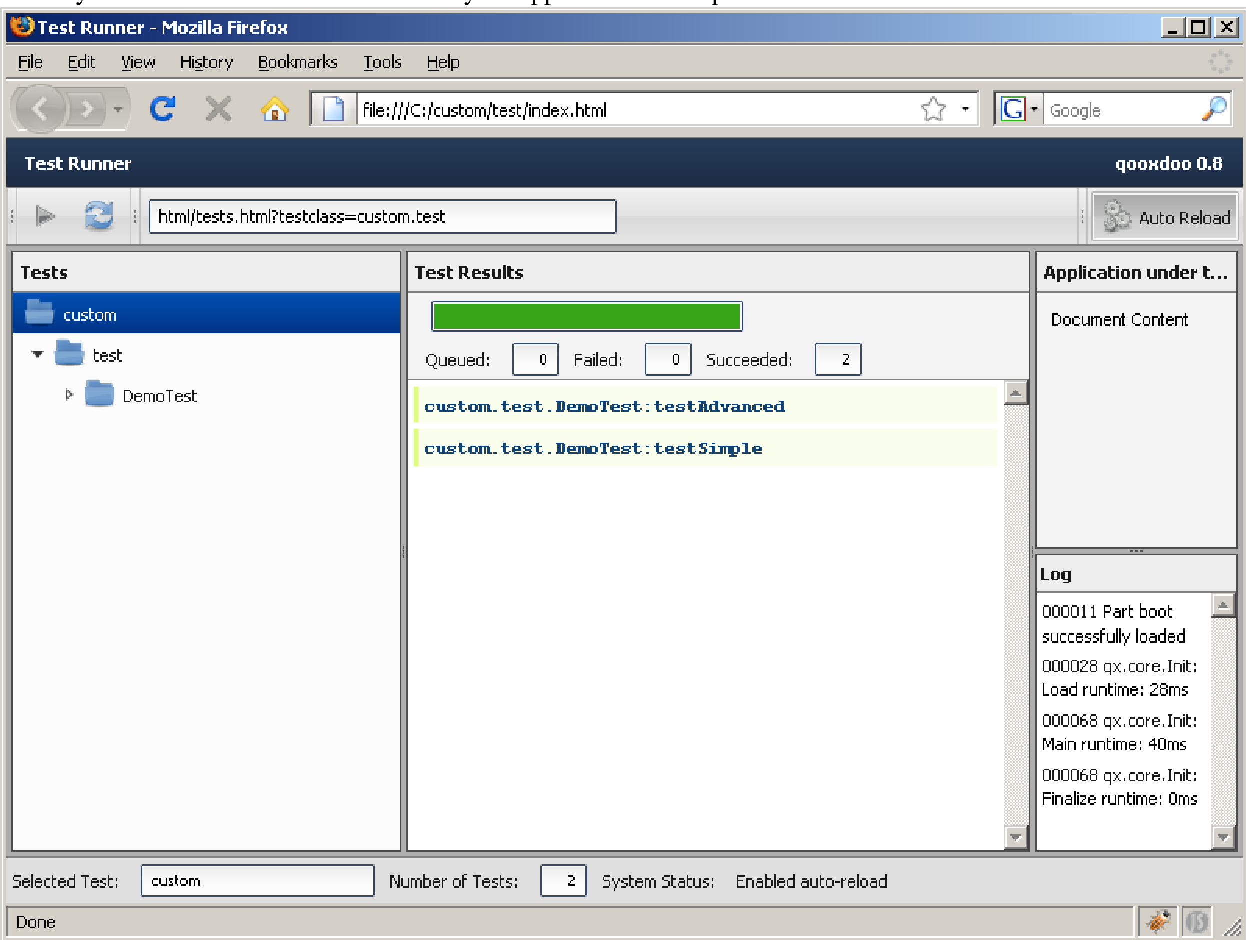


2.2.8 Unit Testing

You might have noticed the `test/DemoTest.js` file in the `source/class` folder of your application. This class demonstrates how to define "unit tests" for your application. qooxdoo comes with its own unit testing framework, it does not require any additional software installation. Simply execute the following command:

```
generate.py test
```

Open `index.html` from the newly created top-level `test` folder in your browser. The "Testrunner" application allows you to select and run the tests under your application namespace:



You may skip the rather advanced topic of unit tests while continuing to extend your custom application code. In case you are interested in test-driven development and creating your own unit tests, please see the corresponding *Unit Testing* documentation.

2.3 Troubleshooting

2.3.1 Python Installation

Python 3.0

Please make sure that you use a regular **Python 2.x** release (v2.5 or above). **Python 3.0 is currently not supported**.

Execute `python -V` in a console to get the installed Python version.

Windows

Making the interpreter available

Note: The following is only required when installing the Windows package from Python.org. When installing the preferred ActivePython this installation step is conveniently handled within its graphical installation wizard.

After your successful *Python installation*, you need to add the installation folder to the so-called `PATH` environment variable, which contains a list of directories that are searched for executables.

Suppose you installed Python to its default location `C:\Python26`, open a Windows command shell (choose menu `Start -> Run...` and type `cmd`). The following command prepends the installation folder to the value of `PATH`, separated by a semicolon:

```
set PATH=C:\Python26;%PATH%
```

When you now execute `python -V`, it should print out its version number.

The modification of the `PATH` variable as described above is only *temporary*. In order not to repeat the command each time you open a new command shell, modify the `PATH` variable permanently: in `Start -> Preferences -> System` choose `Environment variables` under the `Advanced` tab. Edit the system variable `Path` by prepending `C:\Python26;`.

File association

Note: The following is only required when installing the Windows package from Python.org. When installing the preferred ActivePython this installation step is conveniently handled within its graphical installation wizard.

In a standard Python installation on Windows, the `.py` file extension gets associated with the Python interpreter. This allows you to invoke .py files directly. You can check that in the following way at a command prompt:

```
C:\>assoc .py
.py=Python.File
```

If this doesn't work, you can add a file association through `Windows Explorer -> Extras -> Folder Options -> File Types`.

If for any reason you cannot use a file association for .py files, you can still invoke the Python interpreter directly, passing the original command line as arguments. In this case, make sure to provide a path prefix for the script name, even for scripts in the same directory, like so (this will be fixed later):

```
python ./generate.py source
```

Windows Vista

To run qooxdoo's Python-based tools without problems, it is important to have Python installed as an administrator "for all" users.

Administrators installing Python "for all" users on Windows Vista *either* need to be logged in as user `Administrator`, *or* use the runas command, as in:

```
runas /user:Administrator "msiexec /i <path>\<file>.msi"
```

Windows 7

It has been reported that you need to use the PowerShell that comes with Windows 7 for the tools to work properly. The simple command shell doesn't seem to be sufficient. To launch the PowerShell, hit the *WIN+R* keys and enter `powershell`.

Mac OS X

Older Macs (e.g. 10.4) may need an update of the pre-installed Python. See the following comment from the Python on Mac page : "Python comes pre-installed on Mac OS X, but due to Apple's release cycle, it's often one or even two years old. The overwhelming recommendation of the "MacPython" community is to upgrade your Python by downloading and installing a newer version from the Python standard release page."

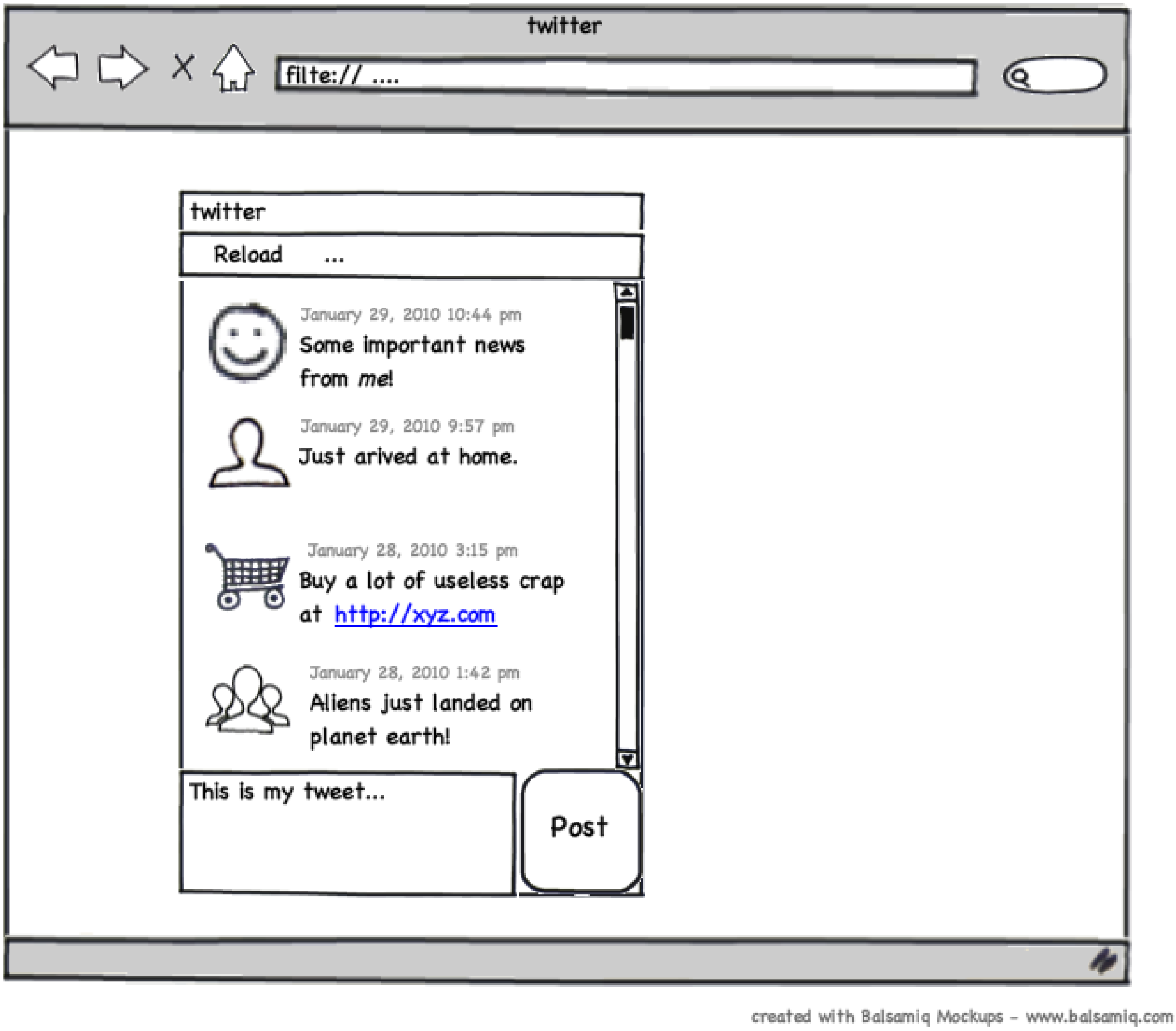
2.4 Tutorials

2.4.1 Tutorial Part 1: The Beginning of a twitter App

The Missing Manual

We have heard it a couple of times: Users are missing a tutorial a bit more complex than the simple *"Hello World" tutorial* we already have. Today, we want to close that gap between the first tutorial and the demo applications included in the framework like the Feedreader.

As you sure have read in the headline, we are building a simple twitter application. twitter is a well known service for posting public short messages and has a good API for accessing data. The following mockup shows you how the application should look like at the end.



If you take a closer look at the mockup, you see a *window* containing a *toolbar*, a *list*, a *text area* and a *button* to post messages. This should cover some common scenarios of a typical qooxdoo application.

In the first part you'll learn how to create a new application and how to build a part of the main UI. But before we get started, be sure you looked at the *"Hello World" tutorial*. We rely on some of the fundamentals explained there.

Getting started

The first step is to get a working qooxdoo application where we can start our development. You should have already have the qooxdoo SDK and know how to use `create-application.py`, so we just create an application called `twitter`.

```
create-application.py -n twitter
```

After that, we should check if everything works as expected. Change the directory to twitter and run `./generate.py source`. Now the skeleton application is ready to run and you can open the index file located in the `source` directory. After that, open the `Application.js` file located in `source/class/twitter/Application.js` with your favorite editor and we are set up for development!

You should see the unchanged skeleton code of the application containing the creation of a button. We don't need that anymore so you can delete it including all the listener stuff.

The first part is to create a Window. As the *Window* contains all the UI controls, we should extend from the qooxdoo Window and add the controls within that class. Adding a new class is as easy as creating a new file. Just create a file parallel to the `Application.js` file named `MainWindow.js`. Now it is time to add some code to that file. We want to create a class so we use the qooxdoo function `qx.Class.define` for that. Add the following lines to your newly created file.

```
qx.Class.define("twitter.MainWindow",
{
  extend : qx.ui.window.Window,

    construct : function()
    {
      this.base(arguments, "twitter")
    }
});
```

We have created our own class extending the qooxdoo Window. In the constructor, we already set the caption of the window, which is the first constructor parameter of the qooxdoo window. So you already have guessed it, `this.base(arguments)` calls the overridden method of the superclass, in this case the constructor. To test the window, we need to create an instance of it in the main application. Add these two lines of code in the Application.js file to create and open the window.

```
var main = new twitter.MainWindow();
      main.open();
```

Now its time to test the whole thing in the browser. But before we can do that, we need to run the generator once more because we added the window class as new dependency. So run `./generate.py source` and open the page in the browser. You should see a window in the top left corner having the name "twitter".

Programming as Configuring

The last task of this tutorial part is to configure the window. Opening the window in the left corner does not look so good, so we should move the window a bit away from the edges of the viewport. To do this add the following line to your application file:

```
main.moveTo(50, 30);
```

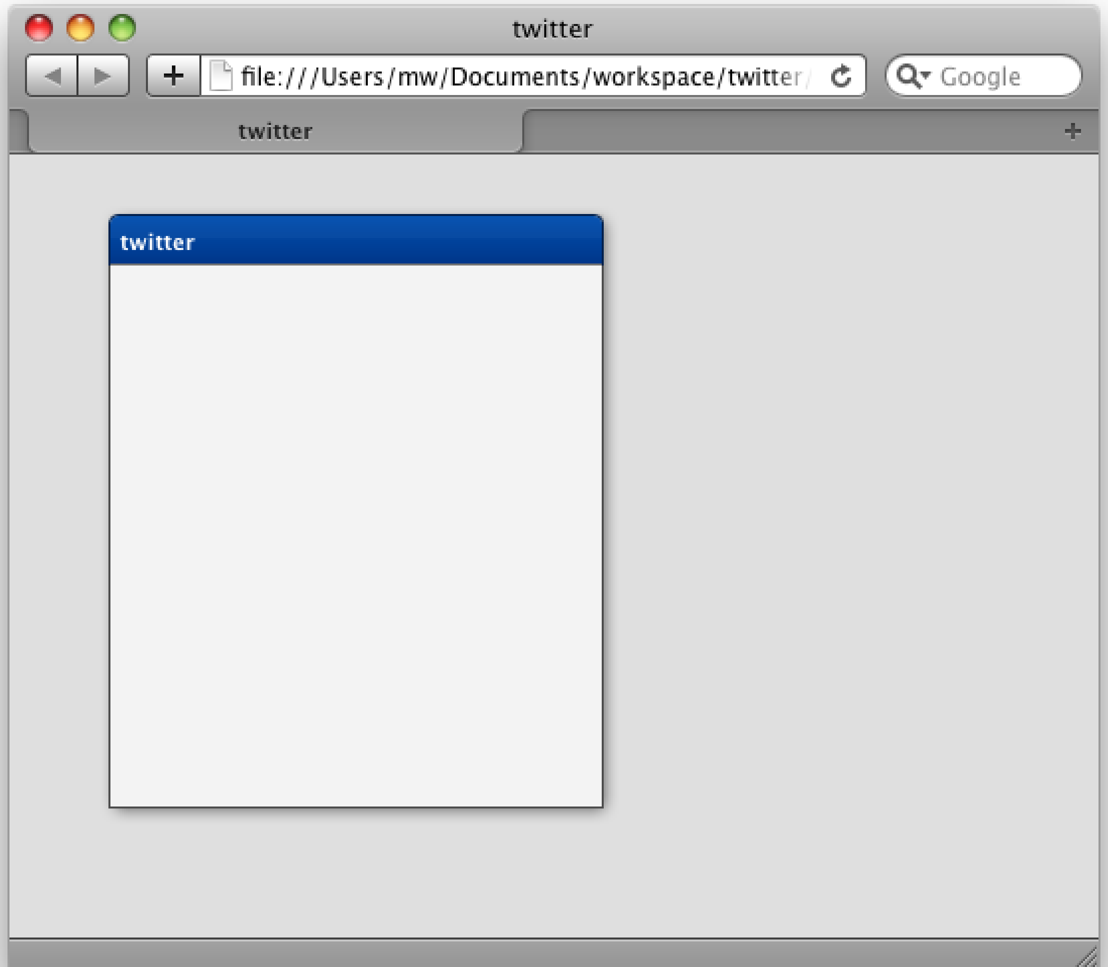
Another thing we should configure are the buttons of the window. The user should not be able to close, minimize nor maximize the window. So we add the following lines of code in our windows constructor.

```
// hide the window buttons
    this.setShowClose(false);
    this.setShowMaximize(false);
    this.setShowMinimize(false);
```

The last thing we could change is the size of the window on startup. Of course the user can resize the window but we should take care of a good looking startup of the application. Changing the size is as easy as hiding the buttons, just tell the window in its constructor:

```
// adjust size
    this.setWidth(250);
    this.setHeight(300);
```

At this point, your application should look like this.

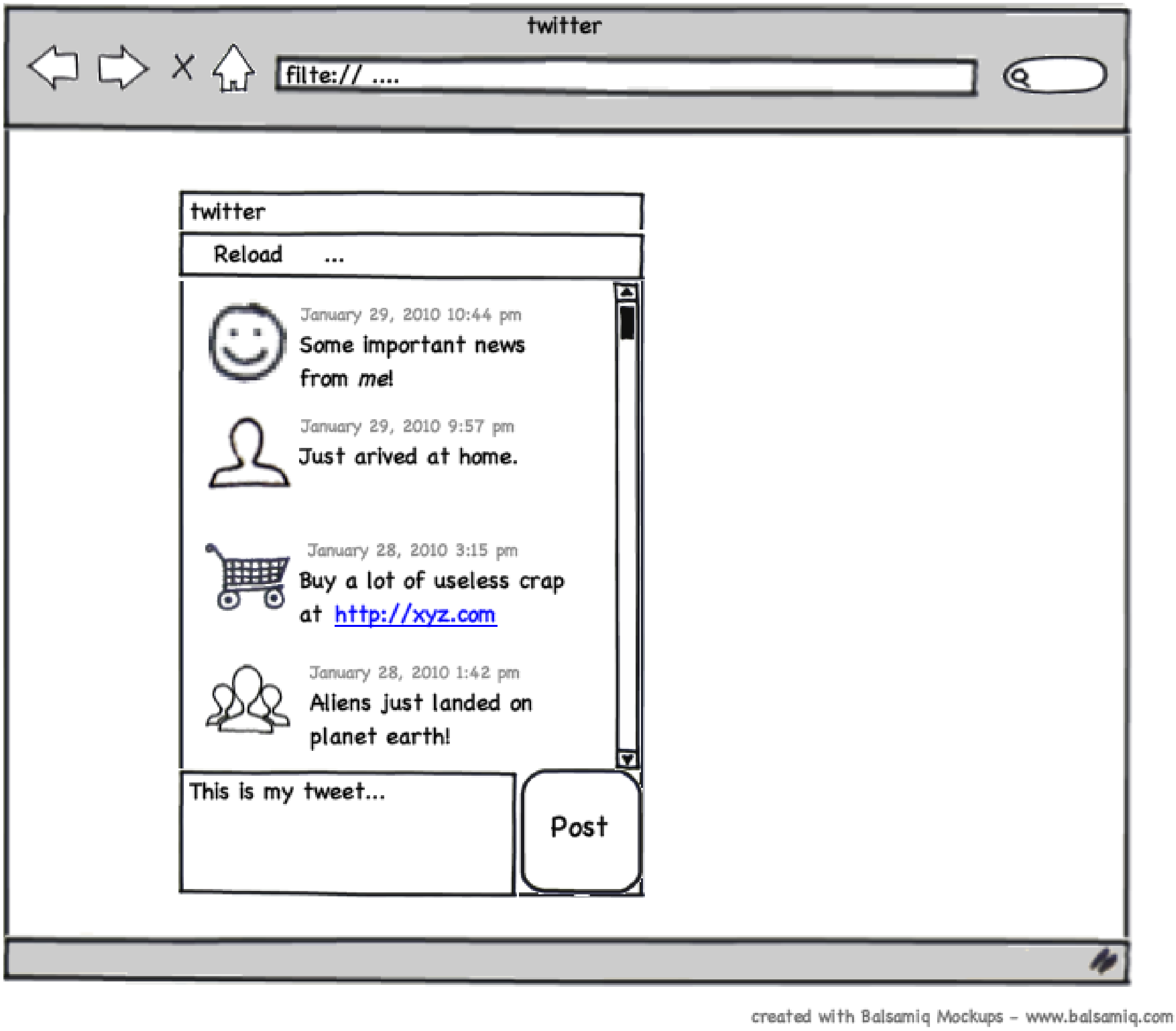


Thats it for the first part. If you want to have the code from the tutorial, take a look at the project on github and just fork the project. The next part of the tutorial will contain the building of the rest of the UI. If you have feedback or want to see something special in further tutorials, just let us know!

2.4.2 Tutorial Part 2: Finishing the UI

In the *first part* of the tutorial, we built a basic window for our target application, a twitter client. In the second part of the tutorial, we want to finish the UI of the application. So lets get started, we got a lot to do!

I hope you remember the layout of the application we are trying to build. If not, here is a little reminder.



The first thing we need to do is to set a layout for our window. You can see that the text area and the button are side by side while all the other elements are ordered vertically. But all elements are aligned in a grid so we should choose a grid layout for that. We can add the grid layout in our own window class. Just add these lines of code in `MainWindow.js`:

```
// add the layout
    var layout = new qx.ui.layout.Grid(0, 0);
    this.setLayout(layout);
```

But a layout without any content is boring so we should add some content to see if it's working. Lets add the first two elements to the window, the *toolbar* and the *list* view.

Layout and Toolbar

First, we need to create the toolbar before we can add it. Creating the toolbar and adding it is straight forward.

```
// toolbar
    var toolbar = new qx.ui.toolbar.ToolBar();
    this.add(toolbar, {row: 0, column: 0});
```

This will add the toolbar to the grid layout of our main window. The only thing you should take care of is the second parameter of .add(). It contains a map with layout properties. You can see the available layout properties in the API of the layout, in this case of the grid layout. Here, we use only the row and column property to tell the layout that this is the element in the first row and column (rows and columns start at index 0, you guessed it).

List and Layout, again

Adding the list should look familiar now.

```
// list
    var list = new qx.ui.form.List();
    this.add(list, {row: 1, column: 0});
```

Now its time to see our work in the browser. But again, we have added new class dependencies so we need to invoke the generator with `./generate.py source`. After that, we can see the result in the browser. I guess it's not the way we like it to be. You cannot see any toolbar, the list has too much padding against the window border and doesn't fit the whole window. That's something we should take care of now.

First, get rid of that padding we don't need. The window object has a default content padding which we just to set to `0`.

```
this.setContentPadding(0);
```

Put that line in your windows constructor and the padding is gone.

Next, we take care of the size of the list. The layout does not know which column(s) or row(s) it should stretch. So we need to tell the layout which one it should use:

```
layout.setRowFlex(1, 1);
    layout.setColumnFlex(0, 1);
```

The first line tells the layout to keep the second row (the row for the list) flexible. The second row does the same for the first column.

The last thing we need to fix was the invisible toolbar. If you know the reason why it's not visible, you sure know how to fix it. It contains not a single element so it won't be visible. Fixing it means adding an element, in our case we just add the reload button. We already know how to create and add widgets so just add the following lines of code.

```
// reload button
    var reloadButton = new qx.ui.toolbar.Button("Reload");
    toolbar.add(reloadButton);
```

Now its time to see if all the fixes work. But be sure to run the generator before you reload the browser page because we added (again) another class (the button). Now everything should look the way we want it to be.

Text Area and Button

After that success, we can got to the next task, adding the text area and "Post" button. This is also straight forward like we have seen in all the other adding scenarios.

```
// textarea
    var textarea = new qx.ui.form.TextArea();
    this.add(textarea, {row: 2, column: 0});

    // post button
    var postButton = new qx.ui.form.Button("Post");
    this.add(postButton, {row: 2, column: 1});
```

This time, we have to add the button in the second column to get the button and the text area aligned horizontally. Its time to test this... again generate and reload.

Like the last time, the result is not quite what we want it to be. The list and toolbar do not fill the whole window. But that's a home-made problem because we extended our grid to two columns by adding the post button. The list and the toolbar need to span both available columns to have the result we want. But that's easy too, add `colSpan: 2` to the layout properties used by adding the list and the toolbar. Your code should look like this:

```
this.add(toolbar, {row: 0, column: 0, colSpan: 2});
    // ...
    this.add(list, {row: 1, column: 0, colSpan: 2});
```

This time, we did not add a new class dependency so we can just reload the index file and see the result.

Breathing Life into the UI

The UI now looks like the one we have seen in the mockup. But how does the UI communicate with the application logic? It's a good idea to decouple the UI from the logic and use events for notifying the behaviour. If you take a look we only have two actions where the UI needs to notify the rest of the application: reloading the tweets and posting a tweet.

These two events we add to our window. Adding events is a two step process. First, we need to declare what kind of event we want to fire. Therefore, we add an events section alongside to the constructor section of the window class definition:

```
events :
  {
    "reload" : "qx.event.type.Event",
    "post"   : "qx.event.type.Data"
  },
```

As you can see in the snippet here, it ends with a comma. It always depends on what position you copy the section if the comma is necessary. Just take care the the class definition is a valid JavaScript object. But now back to the events. The reload event is a plain event which only notifies the receiver to reload. The post event is a data event which contains the data to post to twitter. That's why there are two different types of events used.

Declaring the events is the first step of the process. The second part is firing the events! Let's take a look at the reload event. It needs to be fired when the reload button was triggered (or "was executed" in qooxdoo parlance). The button itself fires an event on execution so we could use this event to fire our own reload event.

```
reloadButton.addListener("execute", function() {
      this.fireEvent("reload");
    }, this);
```

Here we see two things: First, how to add an event listener and second, that firing an event is as easy as a method call. The only parameter to .fireEvent() is the name of the event we have declared in the class definition. Another interesting thing here is the third parameter of the `addListener` call, **this**. It sets the context of the callback function to our window instance, so the this in this.fireEvent() is resolved correctly.

The next case is a bit different but also easy.

```
postButton.addListener("execute", function() {
      this.fireDataEvent("post", textarea.getValue());
    }, this);
```

This time, we call the `fireDataEvent` method to get a data event fired. The second parameter is the data to embed in the event. We simply use the value of the text area. That's it for adding the events. To test both events we add a debug listener for each event in out application code, in the main() method of Application.js:

```
main.addListener("reload", function() {
        this.debug("reload");
      }, this);

      main.addListener("post", function(e) {
        this.debug("post: " + e.getData());
      }, this);
```

You can see in the event listener functions that we use the qooxdoo debugging function `debug`. Now it's time to test the whole UI. Open the index file in a browser you like and see the UI. If you want to see the debugging messages you have to open either a the debugging tool of your chosen browser or use the qooxdoo debugging console. Press F7 to get the qooxdoo console visible.

Finishing Touches

As a last task, we can give the UI some finishing touches. Wouldn't it be nice if the text area had a placeholder text saying you should enter your message here? Easy task!

```
textarea.setPlaceholder("Enter your message here...");
```

Another nice tweak could be a twitter logo in the windows caption bar. Just download this logo from twitter and save it in the `source/resource/twitter` folder of your application. Adding the logo is easy because the window has also a property for an icon, which can be set in the constructor. Adding the reference to the icon in the base call should do the job.

```
this.base(arguments, "twitter", "twitter/t_small-c.png");
```

This time, we added a new reference to an image. Like with class dependencies, we need to run the generator once more. After that, the image should be in the windows caption bar.

Two more minor things are left to finish. First, the button does not look very good. Why don't we just give it a fixed width to fit its height.

```
postButton.setWidth(60);
```

The last task is a bit more complicated than the other tweaks before. As you probably know, twitter messages have a maximum length of 140 characters. So disabling the post button if the entered message has more the 140 characters could help us out in the communication layer. A twitter message with no text at all is also useless and we can disable the post button in that case. To get that we need to know when the text was changed in the text area. Fortunately, the text area has a data event for text changes we can listen to:

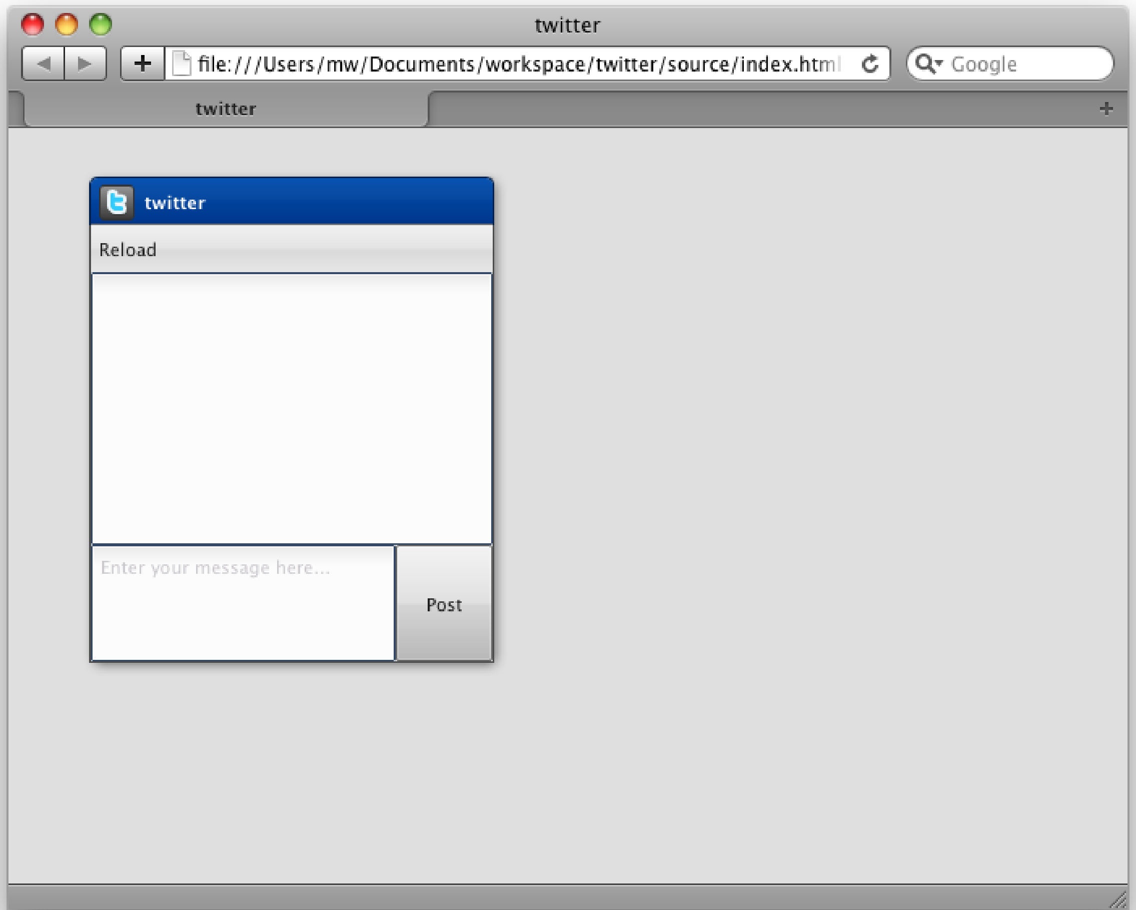
```
textarea.addListener("input", function(e) {
      var value = e.getData();
      postButton.setEnabled(value.length < 140 && value.length > 0);
    }, this);
```

The event handler has only two rows. The first gets the changed text of the text area from the data event. The second row sets the enabled property of the post button if the length of the message is lower than 140 characters and not 0. Some of you might have a bad feeling about this code because the listener is called every time the user adds a character. But that's not a problem because the qooxdoo property system takes care of that. If the value passed into the setter is the same as the existing value, it is ignored and no event is fired.

The last thing we should consider is the startup of the application. The text area is empty but the button is enabled. Disabling the button on startup is the way to go here.

```
postButton.setEnabled(false);
```

Now go back to the browser and test your new tweaks. It should look like this.



That's it for building the UI. Again, if you want to take a look at the code, fork the project on github. Next time we take care of getting the data. If you have feedback on this post, just let us know!

2.4.3 Tutorial Part 3: Time for Communication

After we created the application and the main window in the *first tutorial* part and finished the UI in the *second*, we will build the communication layer today. With that part the application should be ready to use.

Pre-Evaluation

First, we need to specify what's the data we need to transfer. For that, we need to take a look what tasks our application can handle:

1. Show the public twitter timeline.
2. Post a tweet.

So it's clear that we need to fetch the public timeline (that's how it is called by twitter), and we need to post a message to twitter. It's time to take a look at the twitter API so that we know what we need to do to communicate with the service. But keep in mind that we are still on a website so we can't just send some `POST` or `GET` requests due to cross-site scripting restrictions. The one thing we can and should do is take advantage of JSONP. If you have never heard of JSONP, take some time to read the article on ajaxian to get further details.

Creating the Data Access Class

Now, that we know how we want to communicate, we can tackle the first task, fetching the public timeline. twitter offers a JSONP service for that which we can use. Luckily, there is no login process on the server side so we don't need to bother with that in the client. The following URL returns the public timeline wrapped in a JavaScript method call (that's what JSONP is about):

```
http://api.twitter.com/1/statuses/public_timeline.json?callback=methodName
```

Now we know how to get the data from twitter. Its time for us to go back to the qooxdoo code. It is, like in the case of the UI, a good idea to create a separate class for the communication layer. Therefore, we create a class named `TwitterService`. We don't want to inherit from any advanced qooxdoo class so we extend straight from `qx.core.Object`. The code for that class should looks like this:

```
qx.Class.define("twitter.TwitterService",
{
  extend : qx.core.Object,
  members :
  {
  }
});
```

Fetching the Data

As you can see, we omitted the constructor because we don't need it currently. But we already added a members block because we want to add a `method` named `fetchTweets`:

```
fetchTweets : function() {

}
```

Now it's time to get this method working. But how do we load the data in qooxdoo? As it is a JSONP service, we can use the *JSONP data store* contained in the data binding layer of qooxdoo. But we only want to create it once and not every time the method is called. Thats why we save the store as a private instance member and check for the existence of it before we create the store. Just take a look at the method implementation to see how it works.

```
if (this.__store == null) {
  var url = "http://api.twitter.com/1/statuses/public_timeline.json";
  this.__store = new qx.data.store.Jsonp(url, null, "callback");
  // more to do
} else {
  this.__store.reload();
}
```

We already added the code in case the store exists. In that case, we can just invoke a reload. I also mentioned that the instance member should be private. The two underscores (__) *mark the member as private in qooxdoo*. The creation of the store or the reload method call starts the fetching of the data.

But where does the data go? The store has a property called model where the data is available as qooxdoo objects after it finished loading. This is pretty handy because all the data is already wrapped into *qooxdoo objects*! Wait, hold a second, what are *qooxdoo properites*? Properties are a way to store data. You only need to write a *definition for a property* and qooxdoo will generate the mutator and accessor methods for that property. You will see that in just a few moments.

We want the data to be available as a property on our own service object. First, we need to add a property definition to the `TwitterService.js` file. As with the events specification, the property definition goes alongside with the `members` section:

```
properties : {
  tweets : {
    nullable: true,
    event: "changeTweets"
  }
},
```

We named our property tweets and added two configuration keys for it:

- `nullable` describse that the property can be null
- `event` takes the name of the event fired on a change of the property

The real advantage here is the `event` key which tells the qooxdoo property system to fire an event every time the property value changes. This event is mandatory for the whole *data binding* we want to use later. But that's it for setting up a property. You can find all possible property keys *in the documentation*.

Now we need to connect the property of the store with the property of the *twitter service*. That's an easy task with the *single value binding* included in the qooxdoo data binding. Just add the following line after the creation of the data store:

```
this.__store.bind("model", this, "tweets");
```

This line takes care of synchronizing the two properties, the model property of the store and the tweets property of our service object. That means as soon as data is available in the store, the data will also be set as tweets in the twitter service. Thats all we need to do in the twitter service class for fetching the data. Now its time to bring the data to the UI.

Bring the tweets to the UI

For that task we need to go back to our `Application.js` file and create an instance of the new service:

```
var service = new twitter.TwitterService();
```

You remember the debug listener we added in the last tutorial? Now we change the reload listener to fetch the tweets:

```
// reload handling
main.addListener("reload", function() {
  service.fetchTweets();
}, this);
```

Thats the first step of getting the data connected with the UI. We talk the whole time of data in general without even knowing how the data really looks like. Adding the following lines shows a dump of the fetched data in your debugging console.

```
service.addListener("changeTweets", function(e) {
  this.debug(qx.dev.Debug.debugProperties(e.getData()));
}, this);
```

Now it's time for a test. We added a new classes so we need to invoke the generator and load the index file of the application. Hit the reload button of the browser and see the data in your debugging console. The important thing you should see is that the data is an array containing objects holding the items we want to access: the twitter message as `text` and `"user.profile_image_url"` for the users profile picture. After evaluating what we want to use, we can delete the debugging listener.

But how do we connect the available data to the UI? qooxdoo offers *controllers* for connecting data to a list widget. Thats the right thing we need in that case. But we currently can't access the list of the UI. Thats something we need to change.

Switch to the `MainWindow.js` file which implements the view and search for the line where you created the list. We need to implement an accessor for it so its a good idea to store the list as a private instance member:

```
this.__list = new qx.ui.form.List();
```

Of course, we need to change every occurance of the old identifier `list` to the new `this.__list`. Next, we add an accessor method for the list in the members section:

```
getList : function() {
  return this.__list;
}
```

Data Binding Magic

That was an easy one! Now back to the application code in `Application.js`. We need to set up the already mentioned controller. Creating the controller is also straight forward:

```
// create the controller
var controller = new qx.data.controller.List(null, main.getList());
```

The first parameter takes a model we don't have right now so we just set it to null. The second parameter takes the target, the list. Next, we need to specify what the controller should use as label, and what to use as icon:

```
controller.setLabelPath("text");
controller.setIconPath("user.profile_image_url");
```

The last thing we need to do is to connect the data to the controller. For that, we use the already introduced bind method, which every qooxdoo object has:

```
service.bind("tweets", controller, "model");
```

As soon as the tweets are available the controller will know about it and show the data in the list. How about a test of the whole thing right now? You need (again) to tell the generator to build the source version of the application.

After the application has been loaded in the browser, I guess you see nothing until you hit the reload button of the UI. That's one thing we have to fix: Load the tweets at startup. Two other things are not quite the way we want them to be: The tweets get cut off at the end of the list, and the icons can be delivered by twitter in different sizes. So let's fix those three problems.

The first thing is quite easy. We just add a fetch at the end of our application code and that will initiate the whole process of getting the data to the UI:

```
// start the loading on startup
service.fetchTweets();
```

The other two problems have to be configured when creating the items for the list. But wait, we don't create the list items ourselves. Something in the data binding layer is doing that for us and that something is the controller we created. So we need to tell it how to configure the UI elements it is creating. For exactly such scenarios the controller has a way to handle code from the user, a delegate. You can implement the delegate method `configureItem` to manipulate the list item the controller creates:

```
controller.setDelegate({
  configureItem : function(item) {
    item.getChildControl("icon").setWidth(48);
    item.getChildControl("icon").setHeight(48);
```

```
    item.getChildControl("icon").setScale(true);
    item.setRich(true);
  }
});
```

You see that the method has one parameter which is the current UI element which needs to be configured. This item is a list item which stores its icon as a child control you can access with the `getChildControl` method. After that, you can set the width, height and the scaling of the icon. The last line in the configurator set the item to rich, which allows the text to be wrapped. Save your file and give it a try!

Now it should be the way we like it to be. Sure it's not perfect because it has no error handling but that should be good enough for the tutorial.

Posting tweets

As you have seen in the last paragraphs, creating the data access layer is not that hard using qooxdoo's data binding. That is why we want you to implement the rest of the application: Posting of tweets. But I will give you some hints so it does not take that much time for you.

- twitter does only offer an OAuth authentification. Don't make your self too much work by implementing the whole OAuth thing.
- Tweets can be set to twitters web view by just giving a decoded parameter to the URL: http://twitter.com/?status=123

That should be possible for you right now! If you need to take a look at an implementation, you can always take a look at the code on github or fork the project.

That's it for the third part of the tutorial. With this tutorial, the application should be ready and we can continue our next tutorial lines based on this state of the application. As always, if you have any feedback, please let us know!

2.4.4 Tutorial Part 4.1: Form Handling

Note: This tutorial is outdated! twitter changed its API and does not allow basic authentication anymore. Still, the qooxdoo part is valid and worth trying even if you can not access you friends timeline anymore.

In the previous steps of this tutorial, we *laid the groundwork* for a Twitter client application, gave it a *neat UI* and implemented a *communication layer*. One thing this application still lacks is a nice way for users to input their Twitter user name and password in order to post a status update. Fortunately, qooxdoo comes with a *forms API* that takes the pain out of creating form elements and handling user input.

Before we get started, make sure you're working on the version of the Twitter tutorial application tagged with "Step 3" in the GitHub repository. This includes the posting part of the communication layer that we'll be using in this tutorial.

The plan

We want to create a new window with user name and password fields that pops up when the Twitter application starts. The values will be used to retrieve the user's list of Tweets. Seems simple enough, so let's get right down to business.

Creating the login window

We start by creating a new class called twitter.LoginWindow that inherits from qx.ui.window.Window, similar to the MainWindow class from the first part of this tutorial:

```
qx.Class.define("twitter.LoginWindow",
{
  extend : qx.ui.window.Window,
  construct : function()
  {
    this.base(arguments, "Login", "twitter/t_small-c.png");
  }
});
```

The Login window will only contain the form, which takes care of its own layout. So for the window itself, a Basic layout will suffice. We'll also make the window modal:

```
var layout = new qx.ui.layout.Basic();
this.setLayout(layout);
this.setModal(true);
```

Adding the Form

Now it's time to add a form and populate it with a pair of fields:

```
var form = new qx.ui.form.Form();
var username = new qx.ui.form.TextField();
username.setRequired(true);
form.add(username, "Username", null, "username");
var password = new qx.ui.form.PasswordField();
```

```
password.setRequired(true);
form.add(password, "Password", null, "password");
```

Note how the fields are marked as required. This is a simple kind of validation and in this case it's all we need, which is why the third argument for `form.add` is null instead of a validation function. Required fields will be displayed with an asterisk (*) next to their label.

The next step is to add a dash of data binding awesomeness:

```
var controller = new qx.data.controller.Form(null, form);
var model = controller.createModel();
```

Just like in the previous tutorial, we create a controller without a model. Then, we ask the controller to create a model from the form's elements. This model will be used to serialize the form data.

The form still needs a "submit" button, so we'll add one, plus a "cancel" button to close the window:

```
var loginbutton = new qx.ui.form.Button("Login");
form.addButton(loginbutton);
var cancelbutton = new qx.ui.form.Button("Cancel");
form.addButton(cancelbutton);
cancelbutton.addListener("execute", function() {
  this.close();
}, this);
```

That's all the elements we need, let's get them displayed. We'll let one of qooxdoo's built-in form renderer classes worry about the form's layout:

```
var renderer = new qx.ui.form.renderer.Single(form);
this.add(renderer);
```

The renderer is a widget, so we can just add it to the window. In addition to the standard renderers, it's fairly simple to create a cusstom renderer by subclassing qx.ui.form.renderer.AbstractRenderer, though that's outside the scope of this tutorial.

Accessing the form values

Similar to MainWindow, we'll use an event to notify the other parts of our application of changes to the form. As you'll remember, the "event" section is on the same level as the constructor in the class declaration:

```
events : {
  "changeLoginData" : "qx.event.type.Data"
},
```

Then we add a listener to the submit button that retrieves the values from the model object and attaches them to a data event, making sure the form validates, i.e. both fields aren't empty.

```
loginbutton.addListener("execute", function() {
  if (form.validate()) {
    var loginData = {
      username : controller.getModel().getUsername(),
      password : controller.getModel().getPassword()
    };
    this.fireDataEvent("changeLoginData", loginData);
    this.close();
  }
}, this);
```

Tying it all together

Now to integrate the login window with the other parts of the application. Twitter's friends timeline uses .htaccess for authentication so we can add the login details to the request sent by `TwitterService.fetchTweets()`:

```
fetchTweets : function(username, password) {
  if (this.__store == null) {
    var login = "";
    if (username != null) {
      login = username + ":" + password + "@";
    }
    var url = "http://" + login + "twitter.com/statuses/friends_timeline.json";
    this.__store = new qx.data.store.Jsonp(url, null, "callback");
    this.__store.bind("model", this, "tweets");
  } else {
    this.__store.reload();
  }
},
```

All that's left is to show the login window when the application is started and call `fetchTweets` with the information from the `changeLoginData` event. In the main application class, we'll create an instance of twitter.LoginWindow, position it next to the MainWindow and open it:

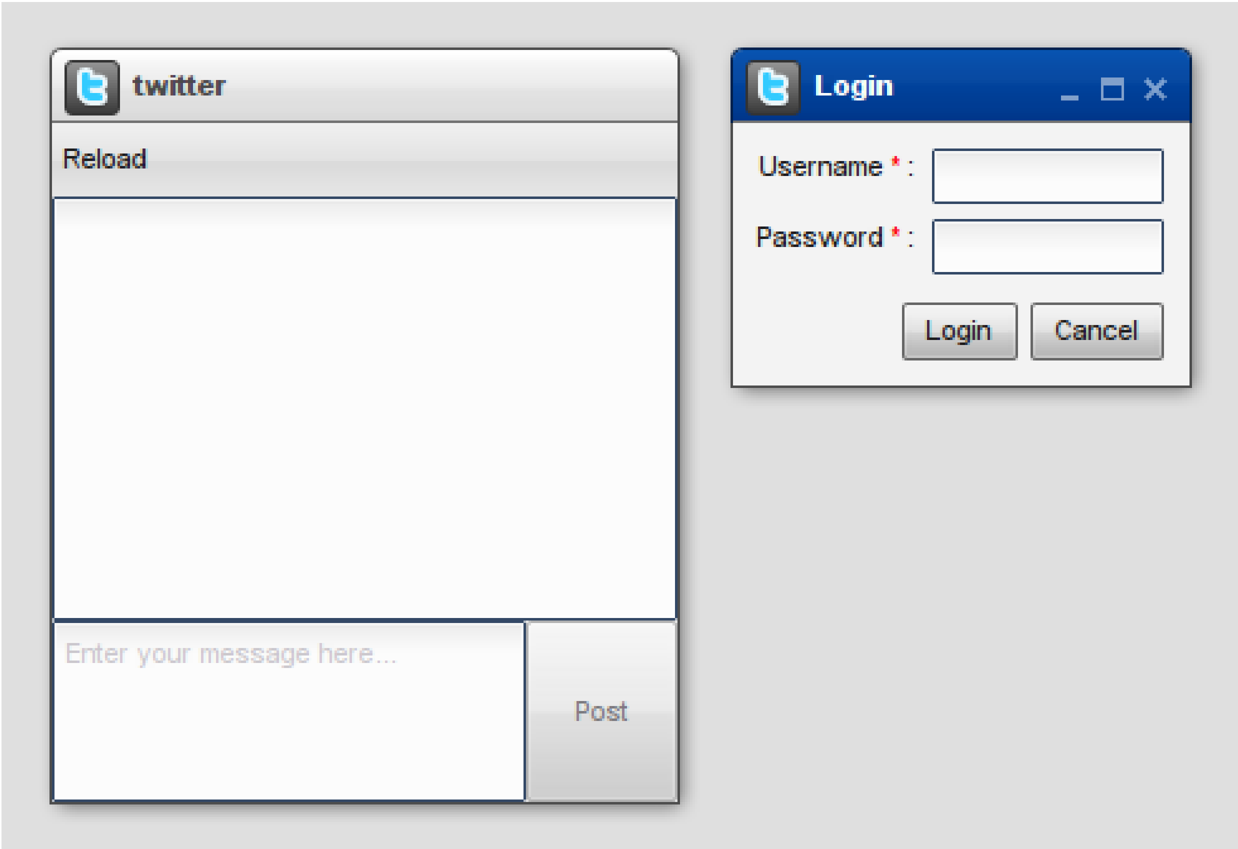
```
this.__loginWindow = new twitter.LoginWindow();
this.__loginWindow.moveTo(320,30);
this.__loginWindow.open();
```

And finally, we'll attach a listener to `changeLoginData`:

```
this.__loginWindow.addListener("changeLoginData", function(ev) {
  var loginData = ev.getData();
  service.fetchTweets(loginData.username, loginData.password);
});
```

Note how all the other calls to `service.fetchTweets` can remain unchanged: By making the login window modal, we've made sure the first call, which creates the store, contains the login data. Any subsequent calls (i.e. after reloading or posting an update) will use the same store so they won't need the login details.

OK, time to run `generate.py source` and load the application in a browser to make sure everything works like it's supposed to.



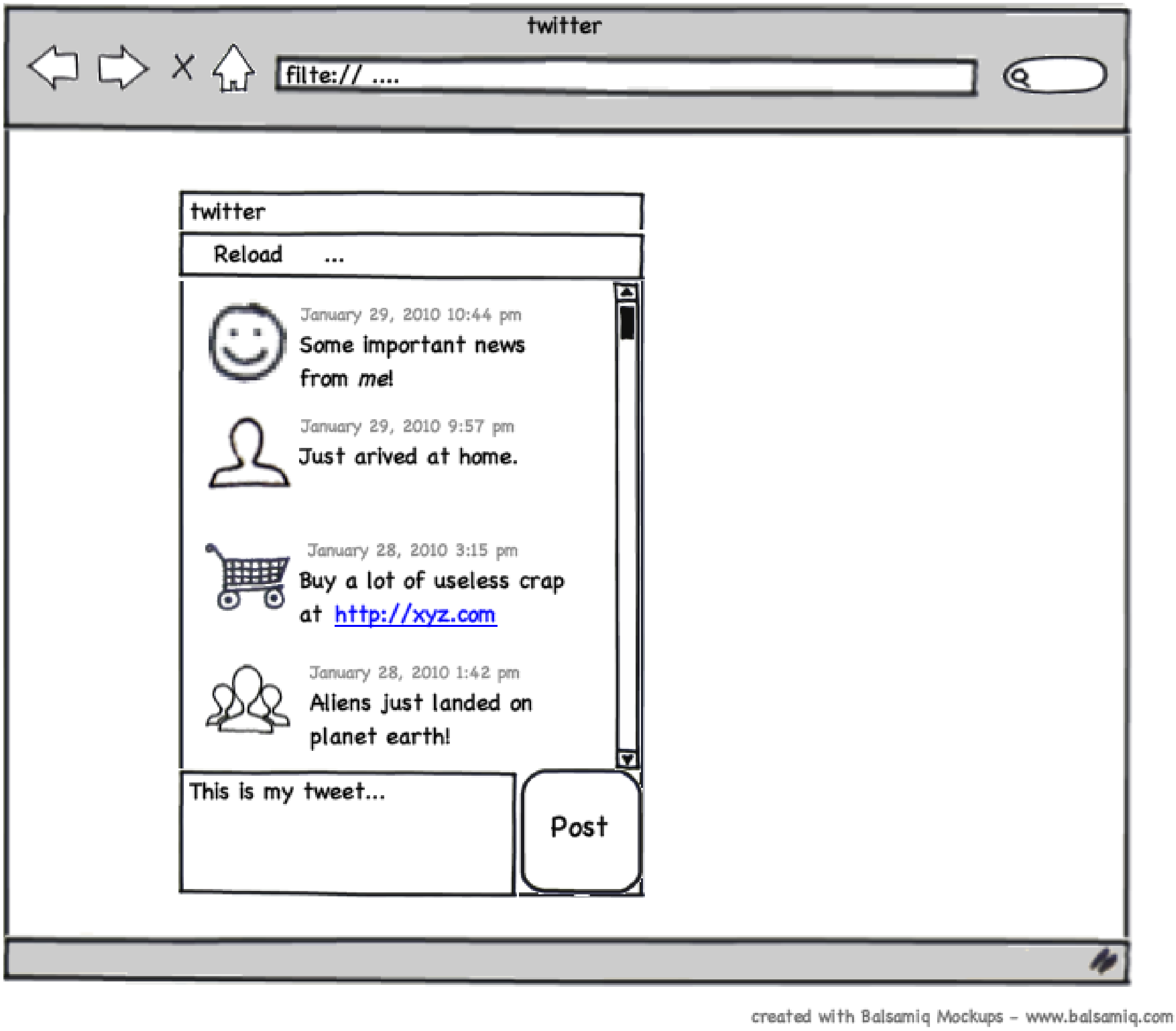
Twitter client application with login window

And that's it for the form handling chapter. As usual, you'll find the tutorial code on GitHub. Watch out for the next chapter, which will focus on developing your own custom widgets.

2.4.5 Tutorial Part 4.2: Custom Widgets

In this tutorial we will deal with how to create a custom widget for our Twitter application. It is necessary that you finished the tutorials part 1 through part 3 to work with this tutorial, but previous knowledge from tutorial 4.1 is not needed.

Do you remember the mockup from tutorial part 1?



You can see that one tweet consists of a photo, a text and a creation date, but at the moment the Twitter application doesn't show the creation date of a tweet. This is because we use the default ListItem to show a tweet and a ListItem can only show an image and/or label. To achieve our goal, we have to create a custom widget which we can use instead of the ListItem.

Note: The code in this tutorial should also work when you haven't completed the 4.1 tutorial because it doesn't depend on the code changes from tutorial 4.1. But if you have any problems to run the tutorial, you can also checkout the code from tutorial 4.1 on github.

The plan

First of all we have to create a custom widget which fulfills our requirements from the mockup. We will achieve this by combining a widget with two labels and one image. Afterwards we have to configure the controller so that it uses our custom widget for the tweets.

Create the custom widget class

You should know how to create a class from the previous tutorials. So please create a class for `twitter.TweetView`, but in our case we need to extend from `qx.ui.core. Widget`.

```
qx.Class.define("twitter.TweetView",
{
  extend : qx.ui.core.Widget,
  include : [qx.ui.form.MModelProperty],

  construct : function() {
    this.base(arguments);
  }
});
```

The attentive reader noticed that we use the `include` key for the first time. `include` is used to include a *mixin* in a class. This is necessary in our case to support Data Binding. Our Twitter application uses it and therefore it is expected that the new widget implements the qx.ui.form.IModel interface. Otherwise the widget can't be used with Data Binding. But fortunately the mixin `qx.ui.form.MModelProperty` already implements it, so we can reuse the implementation.

Define the needed properties

Our widget should show a Tweet as shown in the mockup. To achieve this, we need properties to save the data for a Tweet. Add this definition to the `TweetView` class:

```
properties :
{
  appearance :
  {
    refine : true,
    init : "listitem"
  },

  icon :
  {
    check : "String",
    apply : "_applyIcon",
    nullable : true
  },

  time :
  {
    check : "Date",
    apply : "_applyTime",
    nullable : true
  },

  post :
  {
    check : "String",
    apply : "_applyPost",
    nullable : true
  }
},
```

The properties `icon`, `time` and `post` contain the data from a tweet. In this definition you'll also find a property `appearance`. This property is needed for the theming, it tells the appearance system that the `TweetView` should be styled like the `ListItem`. We could also use a new appearance id, but than we'd have to define an appearance for it and that's not part of this tutorial.

How to define properties was explained in *tutorial part 3*, so we don't repeat it. But we use some unfamiliar keys for definition and I will explain them:

- **check**: check ensures that the incoming value is of this type. But be careful, the check is only done in the source version.
- **apply**: here you can define which method should be called when the value changes.
- **refine**: this is needed when an already defined property should be overridden.
- **init**: defines the initialized value of a property.

Using Child Control

qooxdoo has a special system to realize combined widgets like in our case. This system is called child controls and you can find a detailed documentation in our *manual*.

Okay, back to our problem. To achieve the requirements we need an Image for the photo, a Label for the post and another Label for the creation time. So three widgets, also called sub widgets, are needed for our custom widget. And last but not least the familiar Grid layout for layouting, but that's not created in the child control implementation. We just need to keep it in mind when adding the child control with `_add`.

```
members :
{
  // overridden
  _createChildControlImpl : function(id)
  {
    var control;

    switch(id)
    {
      case "icon":
        control = new qx.ui.basic.Image(this.getIcon());
        control.setAnonymous(true);
        this._add(control, {row: 0, column: 0, rowSpan: 2});
        break;

      case "time":
        control = new qx.ui.basic.Label(this.getTime());
        control.setAnonymous(true);
        this._add(control, {row: 0, column: 1});
        break;

      case "post":
        control = new qx.ui.basic.Label(this.getPost());
        control.setAnonymous(true);
        control.setRich(true);
        this._add(control, {row: 1, column: 1});
        break;
    }

    return control || this.base(arguments, id);
  }
},
```

The child control system has a special method to create sub widgets. The method is called _createChildControlImpl and we override it to create our sub widgets. This method is called from the child control system when it notices that a sub widget is needed but not already created.

In our case:

- **icon**: for the photo
- **time**: for the creation time
- **post**: for the text from the tweet

Dependent on the passed id we create the correct sub widget, configure it and add it to the Grid layout at the right position. If an unknown id is passed, we delegate it to the superclass.

Finishing the constructor

Now i'ts time to finish the constructor.

```
// create a date format like "June 18, 2010 9:31 AM"
this._dateFormat = new qx.util.format.DateFormat(
  qx.locale.Date.getDateFormat("long") + " " +
  qx.locale.Date.getTimeFormat("short")
);
```

The property for the date saves only a date object and our requirement from the mockup describes a spacial format and a simple toString usage is not enough. Therefore we need a special transformation which we can achieve by using DateFormat.

```
// initialize the layout and allow wrap for "post"
var layout = new qx.ui.layout.Grid(4, 2);
layout.setColumnFlex(1, 1);
this._setLayout(layout);
```

Now we create a layout for our custom widget. This should be known from *tutorial part 2*.

```
// create the widgets
this._createChildControl("icon");
this._createChildControl("time");
this._createChildControl("post");
```

Time for our child control implementation. With these lines we trigger the subwidget creation which we implemented before.

Adding the apply methods

We have already defined the properties, but we haven't implemented the needed apply methods for them. So, time to add the missing apply method for the properties to the members section.

```
// property apply
_applyIcon : function(value, old) {
  var icon = this.getChildControl("icon");
  icon.setSource(value);
},

_applyPost : function(value, old) {
  var post = this.getChildControl("post");
  post.setValue(value);
},
```

```
// property apply
_applyTime : function(value, old) {
  var time = this.getChildControl("time");
  time.setValue(this._dateFormat.format(value));
}
```

The apply methods for `icon` and `post` are trivial, we have to ensure that we delegate the value change to the correct widget. To get the correct widget instance we can use the `getChildControl` method and afterwards we can set the value on the widget.

The date, however, needs some extra love. We have to use the DateFormat instance to format the date before we set the value.

Finishing the custom widget

At the end we have to add the attribute `_dateFormat` to the `members` section and a destructor to clean up the created DateFormat instance.

Just add this line at the beginning of the members section:

```
_dateFormat : null,

And the destructor after the members section:

destruct : function() {
  this._dateFormat.dispose();
  this._dateFormat = null;
}
```

Great, now we have finished the custom widget.

Configure the List Controller

At the moment the controller doesn't know that it should use our `TweetView` class. Therefore we have to change the old controller configuration. Search for these lines of code in the `Application.js` file:

```
// create the controller
var controller = new qx.data.controller.List(null, main.getList());
controller.setLabelPath("text");
controller.setIconPath("user.profile_image_url");
controller.setDelegate({
  configureItem : function(item) {
    item.getChildControl("icon").setWidth(48);
    item.getChildControl("icon").setHeight(48);
    item.getChildControl("icon").setScale(true);
    item.setRich(true);
  }
});
```

First of all, remove these two lines:

```
controller.setLabelPath("text");
controller.setIconPath("user.profile_image_url");
```

Now to the delegate, just replace the current delegate with this one:

```
controller.setDelegate({
  createItem : function() {
    return new twitter.TweetView();
  },

  bindItem : function(controller, item, id) {
    controller.bindProperty("text", "post", null, item, id);
    controller.bindProperty("user.profile_image_url", "icon", null, item, id);
    controller.bindProperty("created_at", "time", {
      converter: function(data) {
       if (qx.bom.client.Engine.MSHTML) {
         data = Date.parse(data.replace(/( \+)/, " UTC$1"));
       }
       return new Date(data);
     }
    }, item, id);
  },

  configureItem : function(item) {
    item.getChildControl("icon").setWidth(48);
    item.getChildControl("icon").setHeight(48);
    item.getChildControl("icon").setScale(true);
    item.setMinHeight(52);
  }
});
```

The concept of a delegate should be known from *tutorial part 3*, I will only explain the modifications.

You can see that we added a `createItem` method: With this method we can configure the controller to use our `TweetView` for item creation. The method `bindItem` is used to configure the controller to keep the properties of the model and the widget synchronized. In our case it is important to keep the photo, post and creation date synchronous.

```
controller.bindProperty("text", "post", null, item, id);
```

Let us have a look at the above example. The bindProperty method is responsible for the binding between model and widget. The first parameter is the path from the model, the second is the name of the property in the widget, the third parameter is an options map to do e. g. a conversion, the fourth parameter is the widget and the last is the index.

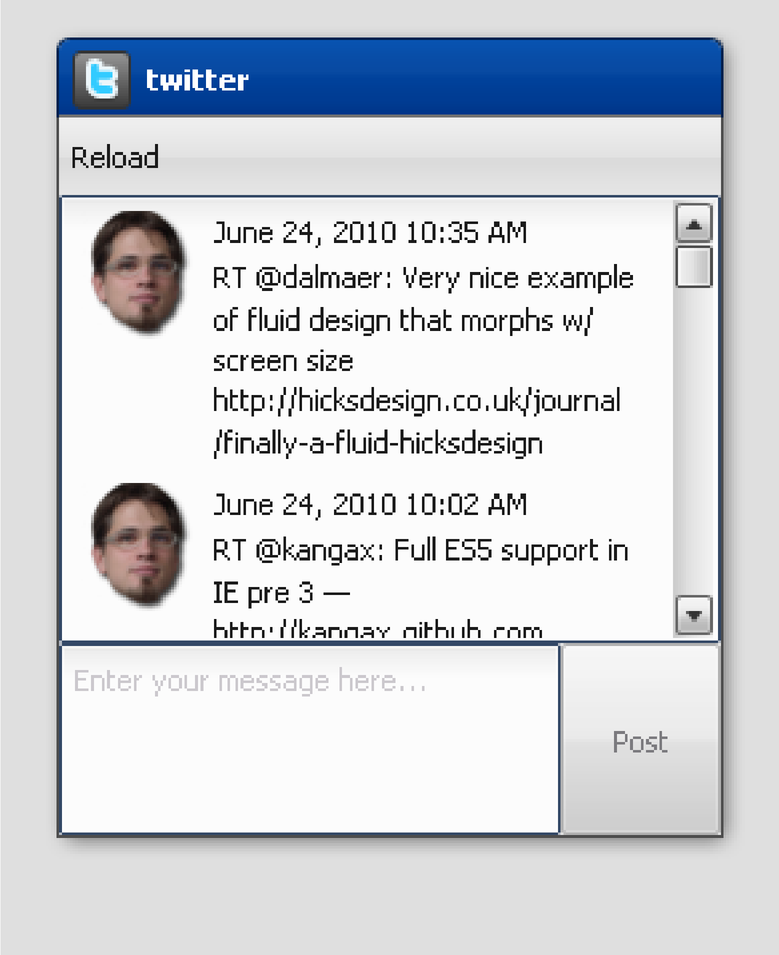
In our case the photo and the post need no conversion because the source data and target data are of the same type. But the creation time needs a conversion because the model contains a String with the UTC time while the widget expects a date object. So we have to convert the data:

```
converter: function(data) {
  if (qx.bom.client.Engine.MSHTML) {
    data = Date.parse(data.replace(/( \+)/, " UTC$1"));
  }
 return new Date(data);
}
```

The converter method creates a date object from the given String. Don't be confused by the if statement. The Twitter model has a format which is not standard UTC format in JavaScript and Internet Explorer has problems parsing the String, therefore a short conversion is needed before the date object can be created.

The `configureItem` method should be known from *tutorial part 3*, there are only some improvements to keep the same behavior as before.

Great, now we've got it! Run `generate.py source` to create the application.



Again, if you want to take a look at the code, fork the project on github.

2.4.6 Tutorial Part 4.3: Translation

We've already covered quite a few of qooxdoo's features to get to this point. In this tutorial, we want to *internationalize* the twitter client. Additionally, we want to add a preferences dialog allowing users to change the language during runtime. Adding a window containing a form should be familiar to you if you've read the *form handling tutorial*

The plan

The first step is to make the application aware of localization. We need to identify all the strings which need to change on a language change. After that, we need to create translations for our initial string set. After that is done, we can add a window containing a radio group with all available language options.

Identifying strings to translate

Now we can benefit from the good design of our application. We put all the view code in our main window which means that's the spot we need to look for strings. Here we can identify the following strings:

```
var reloadButton = new qx.ui.toolbar.Button("Reload");
// ...
```

```
reloadButton.setToolTipText("Reload the tweets.");
// ...
this.__textarea.setPlaceholder("Enter your message here...");
// ...
var postButton = new qx.ui.form.Button("Post");
// ...
postButton.setToolTipText("Post this message on twitter.");
```

qooxdoo offers a handy way to tell both the JavaScript code and the generator which strings need to be translated. Wrapping the strings with `this.tr()` will mark them as translatable strings. That should be an easy task:

```
var reloadButton = new qx.ui.toolbar.Button(this.tr("Reload"));
// ...
reloadButton.setToolTipText(this.tr("Reload the tweets."));
// ...
this.__textarea.setPlaceholder(this.tr("Enter your message here...");)
// ...
var postButton = new qx.ui.form.Button(this.tr("Post"));
// ...
postButton.setToolTipText(this.tr("Post this message on twitter."));
```

Generating the translation files

For the next step, we need to tell the generator what languages we want to support. But why does the generator or the tool chain in general care about that? The tool chain will help us by generating the files necessary for the translation. So we need to edit the config.json file located at the root folder of our application, which is the configuration file for the tool chain. As you can see, this is a plain JSON file which holds some predefined configuration data for the tool chain. You will find a `let` section holding a `LOCALES` key. This key has an array as value holding exactly one locale named `en`, right? In this example, I want to add a translation set for German so I need to add `de` to this array.

```
"LOCALES"      : [ "en" , "de" ],
```

Now we are set up to generate our translation files. For that, just invoke the generator with its translation job.

```
./generate.py translation
```

This will go through all the steps necessary to generate the translation files. But what are translation files anyway? Take a look at the folder `source/translation`. There you'll find the created files which as you'll see end with `.po`. You may be familiar with that file format from GNU gettext which is quite popular.

You should see two files, one for the default language, English (`en.po`), and one for the language you added, in my case German (`de.po`). For now, we just need the file for our alternative language because English is already used in the application so this should work right out of the box. Opening the second file, you'll notice some details about it at the top of the document. The important part starts with the following text.

```
#: twitter/MainWindow.js:30
msgid "Reload"
msgstr ""
```

The first line is a comment, which is a hint containing the class file and line number where the string is used. The second line holds the identifier we used in our application. The third line currently holds an empty string. This is the place where the translation should go for that specific string.

You may have already realized that the rest of the file is a list of blocks similar to this one. Now you should translate all strings and add them in the right spots.

Give it a try

After adding these translations, we should rebuild the application using `./generate.py source` and load it in any browser. If your browser uses the locale you added by default, you should already see the application in the new language. If not, just tell qooxdoo's locale manager to switch the locale using e.g. the Firebug console.

```
qx.locale.Manager.getInstance().setLocale("de"); // or the locale you added
```

If you added a language like German in which most words are longer than in English, you may recognize that we made a mistake in our main window. `postButton.setWidth(60);` may cut off the text in the button because we set the width explicitly. Changing that to `postButton.setMinWidth(60);` will keep the layout flexible for different content sizes.

Adding the preferences window

As you should already be familiar with creating new classes and subclassing a window from the *form handling tutorial*, we won't go into any detail about that again. Just add a new class, subclass the window and override the constructor.

```
qx.Class.define("twitter.SettingsWindow",
{
  extend : qx.ui.window.Window,

  construct : function()
  {
    this.base(arguments, this.tr("Preferences"));
    // ... more to come
  }
});
```

As you can see here, we added another string: The window's caption, which should be translated as well. Keep in mind that you have to use `this.tr()` on every string you add and want to have in your translation file.

For the next step, we need to fill the window with controls. As in the form example, we use a basic layout, a form and some form elements. Add the following line to your constructor.

```
this.setLayout(new qx.ui.layout.Basic());

var form = new qx.ui.form.Form();
var radioGroup = new qx.ui.form.RadioButtonGroup();
form.add(radioGroup, this.tr("Language"));

// TODO: create a radio button for every available locale

var renderer = new qx.ui.form.renderer.Single(form);
this.add(renderer);
```

This code should be familiar to you except for the `RadioButtonGroup`, which is a container for radio buttons. It also makes sure that only one of the buttons is selected at any time. So we don't need to take care of that ourselves. Again, we use a translated string as the label for the radio buttons.

The next step is to access all available locales and the currently set locale. For that, qooxdoo offers a locale manager, as you'll see in the following code part.

```
var localeManager = qx.locale.Manager.getInstance();
var locales = localeManager.getAvailableLocales();
var currentLocale = localeManager.getLocale();
```

It is pretty easy to get this kind of information. You surely know how to continue from here, but before that, I'll show you a little trick. We want to keep the name of the selectable language in the translation file itself. That's a good place to keep that string because otherwise, we would need a mapping from the locale (e.g. en) to its human readable name (e.g. English). Instead we'll, add a special translation key to our application.

```
// mark this for translation (should hold the langauge name)
this.marktr("$$languagename");
```

We will use this key as the label for our radio buttons and then go on, as you would have expected, with a loop for all available locales.

```
// create a radio button for every available locale
for (var i = 0; i < locales.length; i++) {
  var locale = locales[i];
  var languageName = localeManager.translate("$$languagename", [], locale);
  var localeButton = new qx.ui.form.RadioButton(languageName.toString());
  // save the locale as model
  localeButton.setModel(locale);
  radioGroup.add(localeButton);

  // preselect the current locale
  if (currentLocale == locale) {
    localeButton.setValue(true);
  }
};
```

This code contains the rest of the trick. But let's take a detailed look at what we're doing here. The first line of the loop just stores the current locale we want to process. Keep in mind that this is the exact value we need to change the locale later. The second line tells the locale manager to translate the special id we set for the language name using the current locale. This will return a `LocalizedString` which is important to know because these strings update their content on locale switch. But that's not what we want because otherwise, every language will have the same name. Thats why we use the `toString()` method to get the plain string of the current translated value as the label for the new radio button. With that, we exclude the labels for the radio buttons from being translated. The next two tasks are pretty easy: 1) we store the locale as the model of the radio button and 2) we add the radio button to the radio group. Preselecting the currently set locale is really easy as well.

The last thing missing in the window is changing the locale if the user selects a new radio button. For that, we stored the locales in the model property. We can now use the `modelSelection` of the radio button group to react on changes.

```
// get the model selection and listen to its change
radioGroup.getModelSelection().addListener("change", function(e) {
  // selection is the first item of the data array
  var newLocale = radioGroup.getModelSelection().getItem(0);
  localeManager.setLocale(newLocale);
}, this);
```

First, we get the model selection array, which is a data array and has a change event for every change in the array. The new locale is always the first element of the selection array itself, as you can see in the second line. You might have noticed that we need to access the item with a special method instead of the bracket notation normally used with arrays. That's a special method you have to use for data arrays. The third line simply hands the new locale to the manager, which will take care of all the necessary changes.

Accessing the preferences

With that, we are done with the preferences window, but we can't access it yet. We should add a button to the main window's toolbar. Add this code right after where you added the reload button.

```
// spacer
toolbar.addSpacer();

// settings button
var settingsWindow = null;
var settingsButton = new qx.ui.toolbar.Button(this.tr("Preferences"));
toolbar.add(settingsButton);
settingsButton.setToolTipText(this.tr("Change the applications settings."));
settingsButton.addListener("execute", function() {
  if (!settingsWindow) {
    settingsWindow = new twitter.SettingsWindow();
    settingsWindow.moveTo(320,30);
  }
  settingsWindow.open();
}, this);
```

The first thing we do is to add a spacer to attach the preferences button to the right side of the toolbar. This should be the only new thing you haven't seen before, so we won't go into details here.

Final steps

Now we have created some new code containing new strings to translate. Obviously, we need to add translations for these as well. Just run the generator again and let it add the new strings to your `po` files.

```
./generate.py translation
```

Now you can edit the `po` files again and add the new translations. Don't forget to add the translation for the special `$$languagename` key in the english `po` file as well.

After generating the source version of the application again you should be set up for testing and all should run as expected.

I hope you enjoyed this little exercise and gained an idea how easy it is to internationalize an application using qooxdoo's help. As always, you can find the entire code on GitHub. With that said, I want to encourage you to send me pull requests containing alternative translations we could add. It would be interesting to have the twitter app in many different languages. Really looking forward to your feedback and pull requests!

2.5 SDK

2.5.1 Introduction to the SDK

Or *"Everything is a library."*

While the *Hello World* tutorial is geared towards getting you started with your own project, this page walks you through the basic structure of the qooxdoo SDK itself.

There is a page that gives you an overview of the *physical structure* of the SDK. As you can see there the SDK has four main components represented through the subdirectories *application*, *component*, *framework* and *tool*. Three of them, *application*, *component* and *framework* contain (either directly or in further subdirectories) qooxdoo applications or libraries that follow the general scheme for a *qooxdoo application*. In each you will find a *Manifest.json* file which signifies the adherance to the skeleton scheme. They also all contain a *generate.py* script which offers all or a subset of the standard *qooxdoo jobs* that you can run on a library, like *source*, *build*, *test* or *api*.

The fourth component, *tool*, comprises the *tool chain* and its various parts. You shouldn't need to worry about that since you interact with the tool chain through the *generate.py* script or one of the tool/bin scripts like *create-application.py*.

In the SDK's root directory there is - besides *readme.txt* and *license.txt* - an *index.html* that gives you an overview over and access to most of the SDK's applications and components. Just be aware (as mentioned on that page) that all of them need a `generate.py build` first in their respective directories. Only the Apiviewer for the framework is shipped pre-built with the SDK currently and can be invoked immediately.

2.5.2 Framework Structure

When exploring the framework source, the following overview will give you an idea about the file structure of qooxdoo:

application - sample applications (for end users)

- `demobrowser` - for browsing a large number of demos (online)
- `feedreader` - a sample rich internet application (online)
- `portal` - a showcase for low-level features, i.e. without widgets (online)
- `playground` - an interactive playground without the need to install qooxdoo (online)

component - helper applications (used internally)

- `apiviewer` - API reference (for `generate.py api`) (online)
- `skeleton` - blue print for custom applications (for `create-application.py`)
- `testrunner` - unit testing framework (for `generate.py test / test-source`) (online)

framework - main frontend part of the framework

- `source`
  - `class` - JavaScript classes
  - `resource`
    - `qx` - resources need to be namespaced, here it is `qx`
      - `decoration` - images for the decorations, `Modern` and `Classic`
      - `icon` - icon themes that come with qooxdoo, `Oxygen` and `Tango`
      - `static` - other common resources like `blank.gif`
    - `source` - contains original resources
  - `translation` - language-specific data as `po` files

tool - tool chain of the framework

- `bin` - various scripts are located here, most importantly `generator.py`
- `data` - lots of data to be used by different tools, e.g. for localization, migration, etc.
- `pylib` - Python modules used by the platform-independent tool chain

2.5.3 Application Structure

Structural Overview

A qooxdoo application has a well-organized file structure. For an application named `custom`, everything is located within the application folder `custom`. Indentation denotes file system nesting:

- `source` - this folder always exists, as it contains the *development version* of your app

  - `index.html` - usually the only HTML file a qooxdoo application needs. Typically it hardly includes any markup, as the entire qooxdoo application is available as an external JavaScript file
  - `class` - all JavaScript classes
    - `custom` - this is the top-level namespace of your classes, often identical to the application name
  - `resource` - any static resources like images, etc. belong into this folder
    - `custom` - resource handling requires all files to be organized in folders that correspond to namespaces. Typically, the resources of your app are stored in a folder of the same name as the top-level namespace of your application classes
      - `test.png` - sample resource
  - `script` - this folder is created and/or updated for each development version of your app when executing `generate.py source` (or `generate.py source-all`)
    - `custom.js` - this JavaScript file is included from `index.html`. In the `source` version it is a loader script that includes all required files individually.
  - `translation` - if you choose to develop your app for multiple languages, put your translation files into this directory
    - `en.po` - and the other `.po` files for the languages your app supports. The respective locale is used as a file name, e.g. `it.po`, `pt_BR.po`, ...
- `build` - this folder is created and/or updated for each *deployment version* of your app using `generate.py build`
  - `index.html` - identical to the one of the `source` version
  - `script` - contains the generated JavaScript code of your application
    - `custom.js` - this JavaScript file is included from `index.html`. In the `build` version this single file contains all the JavaScript code your application requires, in a compressed and optimized form. If you are developing a large-scale application, you can split it into so-called parts that can be loaded on-demand.
  - `resource` - if your application classes contain appropriate `#asset()` meta information, those resources are automatically copied to this target folder. Your application is then self-contained and may be transferred to an external hosting environment.
- `api` - contains a searchable *API viewer* specific to your application, simply created by `generate.py api`. As it is self-consistent, it may be copied anywhere and be run offline
- `test` - a standalone *Test runner* for unit tests you may create for your app, created by `generate.py test`
- *Manifest.json* - every qooxdoo app has such a Manifest file for some meta information
- *config.json* - configuration file for the build process and all other integrated developer tools
- *generate.py* - you use this platform-independent script for all kinds of tasks and tools, most importantly to generate the development as well as the deployment version of your app

In Other Words

Here is a bit more prose regarding this structure. Of the basic structure, every application/library must contain a *config.json* and a *Manifest.json* file in its top-level directory (In theory, you can deviate from this rule, but it's much easier to stick with it). From this directory, a *source/class* subdirectory is expected, which contains a name space subdirectory and some class files therein. All other subdirectories in the top directory are then created during generator runs ('build', 'api', 'test', ...).

The most important of these sudirectories is of course *source* since it contains your source code. Aside from the *class/<name space>* subdirectory it has to have a *resource* subdir (for icons, style files, flash files, etc.) and a *translation* subdir (for string translation files). All these are mandatory, but might be empty. During a 'generate.py source' a *source/script* directory is created which contains the generator output (basically a Javascript file that references all necessary class files, icons, etc.). This one has to be referenced from the application's index.html (usually *source/index.html*).

The *build* dir (created with 'generate.py build') has a very similar structure as the *source* dir, with *script*, and *resource* subdirs. The main difference is that everything that is necessary for your application to run is copied under this common root, and that the generator output script in *build/script* contains the actual class definitions, not just references to their source files. The *build* dir is therefore self-contained, and doesn't have references that point outside of it.

Create some vanilla skeleton apps with *create-application.py* located in *tool/bin* and look at their initial file structure, to get a feel for it. Tailor the *source/class/<namespace>/Application.js* as the main application class, add further classes to your needs, and let the tool chain take care of the rest. You will have to run `generate.py source` initially and whenever you use further classes in your code. You can try out your app by opening *source/index.html* directly in your browser. You simply reload to see changes in the code. If you are comfortable with that, run a `generate.py build` and open *build/index.html* in your browser. If that is fine, copy the whole `build` tree to your web server.

2.5.4 Manifest.json

Manifest files serve to provide meta information for a library in a structured way. Their syntax is in JSON. They have a more "informal" part (keyed `info`), which is more interesting for human readers, and a technical part (named `provides`) that is used in the processing of generator configurations. Here is a brief sample with all the possible keys:

```
{
  "info" :
  {
    "name" : "Custom Application",

    "summary" : "Custom Application",
    "description" : "This is a skeleton for a custom application with qooxdoo.",

    "keywords" : ["custom"],
    "homepage" : "http://some.homepage.url/",

    "license" : "SomeLicense",
    "authors" :
    [
      {
        "name" : "First Author (uid)",
        "email" : "first.author@some.domain"
      }
    ],

    "version" : "trunk",
    "qooxdoo-versions": ["trunk"]
  },

  "provides" :
  {
    "namespace"   : "custom",
    "encoding"    : "utf-8",
    "class"       : "source/class",
    "resource"    : "source/resource",
    "translation" : "source/translation",
```

```
    "type"         : "application"
  }
}
```

The file paths of the `class`, `resource` and `translation` keys are taken to be relative to the directory of the Manifest file. The `namespace` attribute can be overridden in the importing config file (in the *library* key).

2.5.5 Code Structure

This is how a single source file should look like:

- **UTF-8 encoding**:All source files should be encoded in UTF-8.
- **(optional) Header**:A comment holding author, copyrights, etc.
- **(optional) Compiler Hints**:Can be any number of the following:
  - **#use**(classname) – other class that has to be added to the application; a “run” dependency that has to be available when the current class is actually used (instantiation, method invocation)
  - **#require**(classname) – other class that has to be added to the application before this class; a “load” dependency that has to be available when the current class is loaded into the browser (its code being evaluated)
  - **#ignore**(classname) – unknown global symbol (like a class name) that the compiler should not care about (i.e. you know it will be available in the running application). Ignored symbols will not be warned about. Besides proper class names there are two special symbols you can use:
    - *auto-require* – ignore all *require* dependencies detected by the automatic analysis; they will not be added to the class’ load dependencies
    - *auto-use* – ignore all *use* dependencies detected by the automatic analysis; they will not be added to the class’ run dependencies
  - **#optional**(classname) – this symbol will not be added to either the run or load dependencies of the current class, even if it was detected as a dependency by the automatic analysis
  - **#asset**(resourcepattern) – resources that are used by this class (required if the class uses resources such as icons)
  - **#cldr** – indicates that this class requires CLDR data at runtime
- **Single Definition**: One call to a *define()* method, such as qx.(*Class*|*Theme*|*Interface*|*Mixin*|...).define().

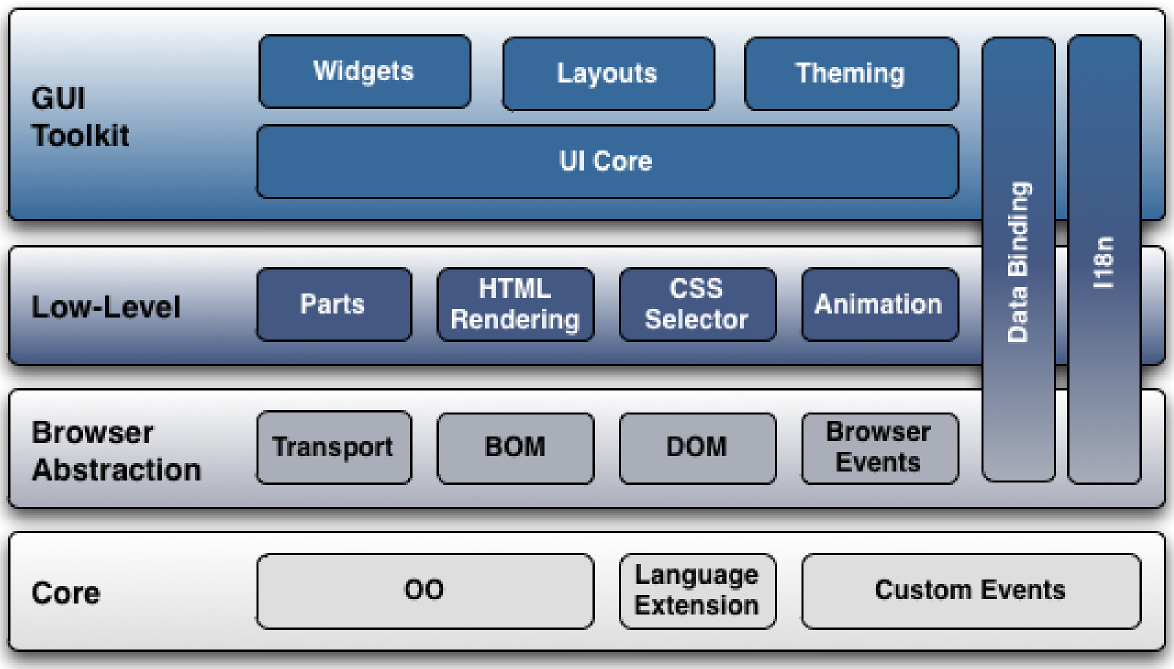
Example:

```
/* ************************************************************************

   Copyright:

   License:

   Authors:

************************************************************************ */

/* ************************************************************************

#require(qx.core.Assert)
#use(qx.log.Logger)
#asset(custom/*)
#ignore(foo)
```

```
************************************************************************ */

qx.Class.define("custom.Application",
{
  ...
});
```

2.5.6 Architecture

UI Architecture



2.5.7 Tools beyond the Python SDK

These are the tools we use that are not self-written, nor part of a vanilla Python 2.x SDK:

Python Modules

(included with the qooxdoo SDK)

Module	License
elementtree	old-style Python (HPND)
graph	MIT
polib	MIT
simplejson	MIT
textile	new BSD
pyparsing	MIT

Other

(not included with the qooxdoo SDK)

Tool	License
ImageMagick	GPL-compat
Sphinx	BSD

CHAPTER

THREE

CORE FRAMEWORK

3.1 Object Orientation

3.1.1 Introduction to Object Orientation

qooxdoo allows you to easily leverage many key concepts of object-oriented programming without bothering about limited native support in JavaScript.

The main actors of qooxdoo OO are:

- **Classes**
- **Interfaces**
- **Mixins**

When trying to get a grip of the framework code, you should probably understand all those three concepts. As a regular application developer you often get by with ignoring interfaces and mixins when starting and just getting familiar with *classes*.

Classes

A "class" is a central concept in most object-oriented languages, and as a programmer you are certainly familiar with it. qooxdoo supports a "closed form" of class declaration, i.e. the entire declaration is provided within a `qx.Class.define(name, config)` statement, where `name` is the fully-qualified class name, and `config` is a configuration map with various keys (or "sections").

There are several types of classes available, which are specified by the `type` key within the `config` map:

- **regular class**: May contain *class* variables/methods (in a `statics` section) and *instance* variables/methods (in a `members` section). An instance of the class can be created using the `new` keyword, so a constructor needs to be given in `construct`.
- **static class**: Only contains class variables and class methods. Often a helper or utility class. Use `type : "static"`.
- **abstract class**: Does not allow an instance to be created. Typically classes derive from it and provide concrete implementations. `type` is `abstract`.
- **singleton**: Not more than a single instance of the class may exists at any time. A static method `getInstance()` returns the instance. Use `type : "singleton"`.

Interfaces

qooxdoo's interfaces are similar to the ones in Java. Similar to the declaration of class they are created by `qx.Interface.define(name, config)`. They specify an "interface" (typically a set of empty methods), that classes must implement.

Mixins

Mixins are a very practical concept that not all programming languages provide. Unlike interfaces, which require a class to provide concrete implementations to fulfill the interface contract, mixins do include code. This code needs to be generic, if it is "mixed into" different existing classes. Mixins usually cover only a single aspect of functionality and therefore tend to be small. They are declared by `qx.Mixin.define(name, config)`.

Inheritance

Like most programming languages qooxdoo only supports single-inheritance for classes, not multiple-inheritance, i.e. a class can only derive directly from a single super class. This is easily modeled by the `extend` key in the class declaration map.

Since a class may implement/include one or many interfaces/mixins, which themselves can extend others, some advanced forms of multiple-inheritance can still be realized.

qooxdoo OO standalone

If you want to use qooxdoo OO layer standalone, take a look at the *qxoo-build* generator job of the framework.

3.1.2 Features of Object Orientation

Class definition

A class is defined by providing its name as a string:

```
qx.Class.define("my.cool.Class");
```

This example only creates a trivial class `my.cool.Class`. A typical class declaration contains OO features like constructor, instance members, static members, etc. This additional information is provided as a second parameter in form of a map. Since the entire class definition is given in `qx.Class.define()`, it is called a "closed form" of class declaration:

```
qx.Class.define("my.cool.Class", {
  // declare constructor, members, ...
});
```

A regular (non-static) class can simply be instantiated using the `new` keyword:

```
var myClass = new my.cool.Class;
```

Inheritance

In order to derive the current class from another class, the reference to the super class is provided by the key `extend`:

```
qx.Class.define("my.great.SuperClass", {
  // I'm the super class
});

qx.Class.define("my.cool.Class", {
  extend : my.great.SuperClass
});
```

Constructor

The constructor of a regular class is provided as a function declaration in key `construct`:

```
qx.Class.define("my.cool.Class",
{
  extend : my.great.SuperClass,
  construct : function() {
    ...
  }
});
```

Static members

Static members (often called "class" members) are also part of the class definition and declared in a map to the `statics` key. Static *methods* are given by providing a function declaration, while all other values declare static *attributes*. Typically they are given in uppercase to distinguish them from instance members:

```
qx.Class.define("my.cool.Class",
{
  statics :
  {
    FOO : VALUE,
    BAR : function() { ... }
  }
});
```

Static members, both methods and attributes, can be accessed by using the fully-qualified class name:

```
my.cool.Class.FOO = 3.141;
my.cool.Class.BAR();
```

Note: You can use static members as constants, but the value can be changed in the run time!!

Instance Members

Similar to static members, instance members are also part of the class definition. For them the `members` key is used:

```
qx.Class.define("my.cool.Class",
{
  members:
  {
    foo : VALUE,
    bar : function() { ... }
  }
});
```

The instance members can be accessed by using an actual instance of a class:

```
var myClass1 = new my.cool.Class;
myClass1.foo = 3.141;
myClass1.bar();
```

Accessing Static Members

Generic form. Requires no updates if class name changes. This code can optionally be optimized for performance in build versions.

```
qx.Class.define("my.cool.Class",
{
  statics : {
    PI : 3.141
  }
  members : {
    circumference : function(radius) {
      return 2 * this.self(arguments).PI * radius;
    }
  }
});
```

Note: For `this.self` to be available, the class must have as a direct or indirect base class `qx.core.Object`.

Note: Static members aren't inherited. For calling a superclass static method, use `this.superclass`, like in this example:

```
qx.Class.define('A', {
  statics: {
     f: function() {}
  }
});

qx.Class.define('B'), {
  extend: A,
  members: {
     e: function() {
        this.superclass.self(arguments).f();
     }
  }
});
```

Static functions can access other static functions directly through the `this` keyword.

Calling the Superclass Constructor

Generic form. Requires no updates if super class (name) changes. This code can optionally be optimized for performance in build versions.

```
qx.Class.define("my.cool.Class",
{
  extend : my.great.SuperClass,
  construct : function(x) {
    this.base(arguments, x);
  }
});
```

Calling the Overridden Superclass Method

Generic form without using `prototype`. Requires no updates if super class (name) changes. This code can optionally be optimized for performance in build versions.

```
qx.Class.define("my.cool.Class",
{
  extend : my.great.SuperClass,
  ...
  members : {
    foo : function(x) {
      this.base(arguments, x);
    }
  }
});
```

Destructor

As a logical match to any existing constructor given by the key `construct`, a destructor is explicitly given by the `destruct` key:

```
qx.Class.define("my.cool.Class",
{
  extend : my.great.SuperClass,
  construct : function() {
    ...
  }
  destruct : function() {
    ...
  }
});
```

Properties

qooxdoo comes with a very powerful feature called dynamic *properties*. A concise declaration of an `age` property may look like the following:

```
qx.Class.define(
...
properties : {
  age: { init: 10, check: "Integer" }
}
...
```

This declaration generates not only a corresponding accessor method `getAge()` and a mutator method `setAge()`, but would allow for many more *features*.

Interfaces

A leading uppercase `I` is used as a naming convention for *interfaces*.

```
qx.Interface.define("my.cool.IInterface");
```

Mixins

Leading uppercase M as a naming convention. A *mixin* can have all the things a class can have, like properties, constructor, destructor and members.

```
qx.Mixin.define("my.cool.MMixin");
```

Attaching mixins to a class

The include key contains either a reference to an single mixin, or an array of multiple mixins:

```
qx.Class.define("my.cool.Class",
{
  include : [my.cool.MMixin, my.other.cool.MMixin]
  ...
});
```

Attaching mixins to an already defined class

```
qx.Class.include(qx.ui.core.Widget, qx.MWidgetExtensions);
```

Access

By the following naming convention. Goal is to be as consistent as possible. During the build process private members can optionally be renamed to random names in order to ensure that they cannot be called from outside the class.

```
publicMember
_protectedMember
__privateMember
```

Static classes

Explicit declaration allows for useful checks during development. For example. construct or members are not allowed for such a purely static class.

```
qx.Class.define("my.cool.Class", {
  type : "static"
});
```

Abstract classes

Declaration allows for useful checks during development and does not require explicit code.

```
qx.Class.define("my.cool.Class", {
  type : "abstract"
});
```

Singletons

Declaration allows for useful checks during development and does not require explicit code. A method `getInstance()` is added to such a singleton class.

```
qx.Class.define("my.cool.Class",
{
  type : "singleton",
  extend :  my.great.SuperClass
});
```

Immediate access to previously defined members

The closed form of the class definition does not allow immediate access to other members, as they are part of the configuration data structure themselves. While it is typically not a feature used very often, it nonetheless needs to be supported by the new class declaration. Instead of some trailing code outside the closed form of the class declaration, an optional `defer` method is called after the other parts of the class definition have been finished. It allows access to all previously declared `statics`, `members` and dynamic `properties`.

Note: If the feature of accessing previously defined members is not absolutely necessary, `defer` **should not be used** in the class definition. It is missing some important capabilities compared to the regular members definition and it cannot take advantage of many crucial features of the build process (documentation, optimization, etc.).

```
qx.Class.define("my.cool.Class",
{
  statics:
  {
    driveLetter : "C"
  },
  defer: function(statics, members, properties)
  {
    statics.drive = statics.driveLetter + ":\\";
    members.whatsTheDrive = function() {
      return "Drive is " + statics.drive;
    };
  }
});
```

Browser specific methods

To maintain the closed form, browser switches on method level is done using *variants*. Since the generator knows about variants it is (optionally) possible to only keep the code for each specific browser and remove the implementation for all other browsers from the code and thus generate highly-optimized browser-specific builds. It is possible to use an logical "or" directly inside a variant key. If none of the keys matches the variant, the "default" key is used:

```
members:
{
  foo: qx.core.Variant.select("qx.client",
  {
    "mshtml|opera": function() {
       // Internet Explorer or Opera
    },
    "default": function() {
       // All other browsers
    }
```

```
  })
}
```

Events

qooxdoo's class definition has a special `events` key. The value of the key is a map, which maps each distinct event name to the name of the event class whose instances are passed to the event listeners. The event system can now (optionally) check whether an event type is supported by the class and issue a warning if an event type is unknown. This ensures that each supported event must be listed in the event map.

```
qx.Class.define("my.eventful.Class",
{
  extend: qx.core.Target,

  events :
  {
    /**  Fired when the widget is clicked. */
    "click": "qx.event.type.MouseEvent"
  }
  ...
})
```

3.1.3 Classes

qooxdoo's class definition is a concise and compact way to define new classes. Due to its closed form the JavaScript code that handles the actual class definition already "knows" all parts of the class at definition time. This allows for many useful checks during development as well as clever optimizations during the build process.

Declaration

Here is the most basic definition of a regular, non-static class `qx.test.Cat`. It has a constructor so that instances can be created. It also needs to extend some existing class, here we take the root class of all qooxdoo classes:

```
qx.Class.define("qx.test.Cat", {
  extend: qx.core.Object,
  construct : function() { /* ... */ }
});
```

As you can see, the `define()` method takes two arguments, the fully-qualified name of the new class, and a configuration map that contains a varying number of predefined keys and their values.

An instance of this class is created and its constructor is called by the usual statement:

```
var kitty = new qx.test.Cat;
```

Members

Members of a class come in two flavors:

- Class members (also called "static" members) are attached to the class itself, not to individual instances
- Instance members are attached to each individual instance of a class

Class Members

A static member of a class can be one of the following:

- Class Variable
- Class Method

In the `Cat` class we may attach a class variable `LEGS` (where uppercase notation is a common coding convention) and a class method `makeSound()`, both in a `statics` section of the class declaration:

```
qx.Class.define("qx.test.Cat", {
  /* ... */
  statics : {
    LEGS: 4,
    makeSound : function() { /* ... */ }
  }
});
```

Accessing those class members involves the fully-qualified class name:

```
var foo = qx.test.Cat.LEGS;
alert(qx.test.Cat.makeSound());
```

Instance Members

An instance member of a class can be one of the following:

- Instance Variable
- Instance Method

They may be defined in the `members` section of the class declaration:

```
qx.Class.define("qx.test.Cat", {
  ...
  members: {
    name : "Kitty",
    getName: function() { return this.name }
  }
});
```

Accessing those members involves an instance of the class:

```
var kitty = new qx.test.Cat;
kitty.name = "Sweetie";
alert(kitty.getName());
```

Primitive Types vs. Reference Types There is a fundamental JavaScript language feature that could lead to problems, if not properly understood. It centers around the different behavior in the assignment of JavaScript's two data types (*primitive types* vs. *reference types*).

Note: Please make sure you understand the following explanation to avoid possible future coding errors.

Primitive types include `Boolean`, `Number`, `String`, `null` and the rather unusual `undefined`. If such a primitive type is assigned to an instance variable in the class declaration, it behaves as if each instance had a copy of that value. They are never shared among instances.

Reference types include all other types, e.g. `Array`, `Function`, `RegExp` and the generic `Object`. As their name suggests, those reference types merely point to the corresponding data value, which is represented by a more complex data structure than the primitive types. If such a reference type is assigned to an instance variable in the class declaration, it behaves as if each instance just pointed to the complex data structure. All instances share the same value, unless the corresponding instance variable is assigned a different value.

Example: If an instance variable was assigned an array in the class declaration, any instance of the class could (knowingly or unknowingly) manipulate this array in such a way that each instance would be affected by the changes. Such a manipulation could be pushing a new item into the array or changing the value of a certain array item. All instances would share the array.

You have to be careful when using complex data types in the class declaration, because they are shared by default:

```
members:
{
  foo: [1, 2, 4]   // all instances would start to share this data structure
}
```

If you do *not* want that instances share the same data, you should defer the actual initialization into the constructor:

```
construct: function()
{
  this.foo = [1, 2, 4];   // each instance would get assigned its own data structure
},
members:
{
  foo: null   // to be initialized in the constructor
}
```

Access

In many object-oriented classes a concept exists that is referred to as “access” or “visibility” of members (well, or even classes, etc.). Based on the well-known access modifiers of Java, the following three types exist for qooxdoo members:

- *public*: To be accessed from any class/instance
- *protected*: To be accessed only from derived classes or their instances
- *private*: To be accessed only from the defining class/instance

Unfortunately, JavaScript is very limited in *enforcing* those protection mechanisms. Therefore, the following coding convention is to be used to declare the access type of members:

- *public*: members may *not* start with an underscore
- *protected*: members start with a single underscore _
- *private*: members start with a double underscore __

There are some possibilities to enforce or at least check the various degrees of accessibility:

- automatic renaming of private members in the build version could trigger errors when testing the final app
- checking instance of `this` in protected methods
- ...

Special Types of Classes

Besides a "regular" class there is built-in support for the following special types:

Static Classes A static class is not instantiated and only contains static members. Setting its type to `static` makes sure only such static members, no constructor and so on are given in the class definition. Otherwise error messages are presented to the developer:

```
qx.Class.define("qx.test.Cat", {
  type : "static"
  ...
});
```

Abstract Classes An abstract class may not be instantiated. It merely serves as a superclass that needs to be derived from. Concrete classes (or concrete members of such derived classes) contain the actual implementation of the abstract members. If an abstract class is to be instantiated, an error message is presented to the developer.

```
qx.Class.define("qx.test.Cat", {
  type : "abstract"
  ...
});
```

Singletons The singleton design pattern makes sure, only a single instance of a class may be created. Every time an instance is requested, either the already created instance is returned or, if no instance is available yet, a new one is created and returned. Requesting the instance of such a singleton class is done by using the `getInstance()` method.

```
qx.Class.define("qx.test.Cat", {
  type : "singleton"
  ...
});
```

Inheritance

Single Inheritance

JavaScript supports the concept of single inheritance. It does not support (true) multiple inheritance like C++. Most people agree on the fact that such a concept tends to be very complex and error-prone. There are other ways to shoot you in the foot. qooxdoo only allows for single inheritance as well:

```
qx.Class.define("qx.test.Cat", {
  extend: qx.test.Animal
});
```

Multiple Inheritance

Not supported. There are more practical and less error-prone solutions that allow for typical features of multiple inheritance: Interfaces and Mixins (see below).

Polymorphism (Overriding)

qooxdoo does, of course, allow for polymorphism, that is most easily seen in the ability to override methods in derived classes.

Calling the Superclass Constructor It is hard to come up with an appealing syntax and efficient implementation for calling the superclass constructor from the constructor of a derived class. You simply cannot top Java's `super()` here. At least there is some generic way that does not involve to use the superclass name explicitly:

```
qx.Class.define("qx.test.Cat", {
  extend: qx.test.Animal,
  construct: function(x) {
    this.base(arguments, x);
  }
});
```

Unfortunately, to mimic a `super()` call the special variable `arguments` is needed, which in JavaScript allows a context-independent access to the actual function. Don't get confused by its name, you would list your own arguments just afterwards (like the `x` in the example above).

`this.base(arguments, x)` is internally mapped to `arguments.callee.base.call(this, x)` (The *.base* property is maintained for every method through qooxdoo's class system). The latter form can be handled by JavaScript natively, which means it is quite efficient. As an optimization during the build process such a rewrite is done automatically for your deployable application.

Calling an Overridden Method Calling an overridden superclass method from within the overriding method (i.e. both methods have the same name) is similar to calling the superclass constructor:

```
qx.Class.define("qx.test.Cat", {
  extend: qx.test.Animal,
  members: {
    makeSound : function() {
      this.base(arguments);
    }
  }
});
```

Calling the Superclass Method or Constructor with all parameters This variant allows to pass all the parameters (unmodified):

```
qx.Class.define("qx.test.Animal", {
  members: {
    makeSound : function(howManyTimes) {
       ....
    }
  }
});

qx.Class.define("qx.test.Cat", {
  extend: qx.test.Animal,
  members: {
    makeSound : function() {
      this.debug("I'm a cat");
      /* howManyTimes or any other parameter are passed.  We don't need to know how many parameters a
      arguments.callee.base.apply(this, arguments);
```

```
    }
  }
});
```

Calling another Static Method Here is an example for calling a static member without using a fully-qualified class name (compare to `this.base(arguments)` above):

```
qx.Class.define("qx.test.Cat", {
  extend: qx.test.Animal,
  statics : {
    someStaticMethod : function(x) {
      ...
    }
  },
  members: {
    makeSound : function(x) {
      this.constructor.someStaticMethod(x);
    }
  }
});
```

The syntax for accessing static variables simply is `this.constructor.someStaticVar`. Please note, for `this.constructor` to be available, the class must be a derived class of `qx.core.Object`, which is usually the case for regular, non-static classes.

Instead of `this.constructor` you can also use the alternative syntax `this.self(arguments)`.

In purely static classes for calling a static method from another static method, you can directly use the `this` keyword, e.g. `this.someStaticMethod(x)`.

Usage of Interfaces and Mixins

Implementing an Interface

The class system supports *Interfaces*. The implementation is based on the feature set of Java interfaces. Most relevant features of Java-like interfaces are supported. A class can define which interface or multiple interfaces it implements by using the `implement` key:

```
qx.Class.define("qx.test.Cat", {
  implement : [qx.test.IPet, qx.test.IFoo]
});
```

Including a Mixin

Unlike interfaces, *Mixins* do contain concrete implementations of methods. They borrow some ideas from Ruby and similar scripting languages.

Features:

- Add mixins to the definition of a class: All members of the mixin are added to the class definition.
- Add a mixin to a class after the class is defined. Enhances the functionality but is not allowed to overwrite existing members.
- Patch existing classes. Change the implementation of existing methods. Should normally be avoided but, as some projects may need to patch qooxdoo, we better define a clean way to do so.

The concrete implementations of mixins are used in a class through the key `include`:

```
qx.Class.define("qx.test.Cat", {
  include : [qx.test.MPet, qx.test.MSleep]
});
```

Summary

Configuration

Key	Type	Description
type	String	Type of the class. Valid types are `abstract`, `static` and `singleton`. If unset it defaults to a regular non-static class.
extend	Class	The super class the current class inherits from.
implement	Interface | Interface[]	Single interface or array of interfaces the class implements.
include	Mixin | Mixin[]	Single mixin or array of mixins, which will be merged into the class.
construct	Function	The constructor of the class.
statics	Map	Map of static members of the class.
properties	Map	Map of property definitions. For a description of the format of a property definition see qx.core.Property.
members	Map	Map of instance members of the class.
settings	Map	Map of settings for this class. For a description of the format of a setting see qx.core.Setting.
variants	Map	Map of settings for this class. For a description of the format of a setting see qx.core.Variant.
events	Map	Map of events the class fires. The keys are the names of the events and the values are the corresponding event type class names.
defer	Function	Function that is called at the end of processing the class declaration. It allows access to the declared statics, members and properties.
destruct	Function	The destructor of the class.

References

- *Class Declaration Quick Ref* - a quick syntax overview
- API Documentation for Class

3.1.4 Interfaces

qooxdoo supports Java like interfaces.

Interface definitions look very similar to normal class definitions.

Example:

```
qx.Interface.define("qx.test.ISample",
 {
   extend: [SuperInterfaces],
```

```
    properties: {"color": {}, "name": {} },

    members:
    {
      meth1: function() {},
      meth2: function(a, b) {
        this.assertArgumentsCount(arguments, 2, 2);
      },
      meth3: function(c) {
        this.assertInterface(c, qx.some.IInterface);
      }
    },

    statics:
    {
      PI : 3.14
    },

    events :
    {
      keydown : "qx.event.type.KeyEvent"
    }
});
```

Definition

Interfaces are declared using `qx.Interface.define`. Interface names start by convention with an `I` (uppercase "i"). They can inherit from other interfaces using the `extend` key. Multiple inheritance of interfaces is supported.

Properties

Properties in interfaces state that each class implementing this interface must have a property of the given name. The *property definition* is not evaluated and may be empty.

Members

The member section of the interface lists all member functions which must be implemented. The function body is used as a precondition of the implementation. By implementing an interface the qooxdoo class definition automatically wraps all methods required by the interface. Before the actual implementation is called, the precondition of the interface is called with the same arguments. The precondition should raise an exception if the arguments are don't meet the expectations. Usually the methods defined in qx.core.MAssert are used to check the incoming parameters.

Statics

Statics behave exactly like statics defined in mixins and qooxdoo classes, with the different that only constants are allowed. They are accessible through their fully-qualified name. For example, the static varaiable `PI` could be used like this:

```
var a = 2 * qx.test.ISample.PI * (r*r);
```

Events

Each event defined in the interface must be declared in the implementing classes. The syntax matches the `events` key of the class declaration.

Implementation

With `implement` key of the class declaration, a list of interfaces can be listed, which the class implements. The class must implement all properties, members and events declared in the interfaces. Otherwise a runtime error will be thrown.

Example:

```
qx.Class.define("qx.test.Sample",
 {
   implement: [qx.test.ISample],

   properties: {
     "color": { check: "color"},
     "name": { check: "String"}
   },

   members:
   {
     meth1: function() { return 42; },
     meth2: function(a, b) { return a+b },
     meth3: function(c) { c.foo() }
   }

   events :
   {
     keydown : "qx.event.type.KeyEvent"
   }

});
```

Validation

`qx.Class` contains several static methods to check, whether a class or an object implements an interface:

- `qx.Class.hasInterface()`: Whether a given class or any of its superclasses includes a given interface.
- `qx.Class.implementsInterface()`: Checks whether all methods defined in the interface are implemented in the class. The class does not need to implement the interface explicitly.

It is further possible to use interfaces as property checks.

Summary

Configuration

Key	Type	Description
extend	Interface | Interface[]	Single interface or array of interfaces this interface inherits from.
members	Map	Map of members of the interface.
statics	Map	Map of statics of the interface. The statics will not get copied into the target class. This is the same behavior as statics in mixins.
properties	Map	Map of properties and their definitions.
events	Map	Map of event names and the corresponding event class name.

References

- *Interfaces Quick Ref* - a syntax quick reference for interfaces
- API Documentation for Interface

3.1.5 Mixins

Mixins are collections of code and variables, which can be merged into other classes. They are similar to classes but can not be instantiated. Unlike interfaces they do contain implementation code. Typically they are made up of only a few members that allow for a generic implementation of some very specific functionality.

Mixins are used to share functionality without using inheritance and to extend/patch the functionality of existing classes.

Definition

Example:

```
qx.Mixin.define("name",
{
  include: [SuperMixins],

  properties: {
    "tabIndex": {check: "Number", init: -1}
  },

  members:
  {
    prop1: "foo",
    meth1: function() {},
    meth2: function() {}
  }
});
```

Usage

Here a short example to see, how to use mixins (MMixinA, MMixinB) with a class (ClassC).

The first mixin:

```
qx.Mixin.define("demo.MMixinA",
{
  properties: {
    "propertyA":
    {
      check: "String",
      init: "Hello, I'm property A!\n"
    }
  },

  members:
  {
    methodA: function() {
      return "Hello, I'm method A!\n";
    }
  }
});
```

The second mixin:

```
qx.Mixin.define("demo.MMixinB",
{
  properties: {
    "propertyB":
    {
      check: "String",
      init: "Hello, I'm property B!\n"
    }
  },

  members:
  {
    methodB: function() {
      return "Hello, I'm method B!\n";
    }
  }
});
```

The usage in the class:

```
qx.Class.define("demo.ClassC",
{
  extend : qx.core.Object,

  include : [demo1.MMixinA, demo1.MMixinB],

  members :
  {
    methodC : function() {
      return this.getPropertyA() + this.methodA()
        + this.getPropertyB() + this.methodB()
        + "Nice to meet you. Thanks for your help!";
    }
```

```
  }
});
```

The result is when calling the method `methodC()` of `ClassC`:

```
var classC = new demo.ClassC;
var result = classC .methodC();
/*
 * Result:
 * Hello, I'm property A!
 * Hello, I'm method A!
 * Hello, I'm property B!
 * Hello, I'm method B!
 * Nice to meet you. Thanks for your help!
 */
```

Summary

Configuration

Key	Type	Description
include	Mixin or Mixin[]	Single mixin or array of mixins, which will be merged into the mixin.
construct	Function	An optional mixin constructor. It is called when instantiating a class that includes this mixin.
destruct	Function	An optional mixin destructor.
statics	Map	Map of static members of the mixin. The statics will not get copied into the target class. They remain accessible from the mixin. This is the same behaviour as for statics in interfaces
members	Map	Map of members of the mixin.
properties	Map	Map of *property definitions*.
events	Map	Map of events the mixin fires. The keys are the names of the events and the values are the corresponding event type classes.

References

- *Mixin Quick Ref* - a quick syntax reference for mixins
- API Documentation for Mixin

3.2 Properties

3.2.1 Introduction to Properties

qooxdoo comes with its own convenient and sophisticated property management system. In order to understand its power we will first take a look at the ordinary property handling in plain JavaScript first.

Ordinary Property Handling

Let's say we have a property `width` for an object `obj`.

As is a good practice in regular high-level programming languages you should not access object properties directly:

```
// NOT RECOMMENDED: direct access to properties
obj.width = 200;  // setting a value
var w = obj.width;  // getting the current value
```

Instead you should work with properties only through so-called *accessor methods* ("getters") and *mutator methods* ("setters"):

```
// direct access is no good practice
obj.setWidth(200);  // setting a value
var w = obj.getWidth();  // getting the current value
```

Of course, directly accessing properties may be faster because no indirection by a function call is needed. Nonetheless, in practice this does not outweight the disadvantages. Direct access to properties does not hide internal implementation details and is a less maintainable solution (Well, you don't program web applications in assembler code, do you?).

A typical implementation of the accessor and mutator methods would look like the following, where those instance methods are declared in the `members` section of the class definition:

```
// ordinary example #1
members:
{
  getWidth : function() {
    return this._width;
  },

  setWidth : function(width)
  {
    this._width = width;
    return width;
  }
}
```

Something that is very familiar to the typical programmer of Java or any other comparable language. Still, it is not very convenient. Even this trivial implementation of only the basic feature requires a lot of keystrokes. More advanced features like type checks, performance optimizations, firing events for value changes, etc. need to be coded by hand. An improved version of the setter could read:

```
// ordinary example #2
members:
{
  setWidth : function(width)
  {
    if (typeof width != "number") {
      // Type check: Make sure it is a valid number
      throw new Error("Invalid value: Need a valid integer value: " + width);
    };

    if (this._width != width)
    {
      // Optimization: Only set value, if different from the existing value
      this._width = width;

      // User code that should be run for the new value
      this.setStyleProperty("width", width+ "px");
```

```
    };

    return width;
  }
}
```

Large part of the code found here is for managing the validation and storage of the incoming data. The property-specific user code is rather short.

qooxdoo Property Handling

Let's see how the above example can be written using qooxdoo's property implementation. The property itself is declared in the `properties` section of the class definition. Only if some property-specific code needs to be run in the setter, an additional `apply` method has to be given:

```
// qooxdoo version of ordinary example #2
properties : {
  width : { check : "Number", apply : "applyWidth" }
}

members :
{
  applyWidth : function(value) {
    this.setStyleProperty("width", value + "px");
  }
}
```

Compare that to the lengthy code of the ordinary code example above! Much shorter and nicer, also by objective means. And it almost only contains the "real code".

The apply method may optionally be defined for each property you add to your class. As soon as you define a key "apply" in your property declaration map the method gets automatically called on each property modification (but not during initial initialization). If you do not define an apply method, the property just handles the fundamental storage of your data and its disposal.

Despite needing much less explicit code (keep in mind, for *every* property), it actually contains at least as many features as the hand-tuned code: The type of the property is checked automatically (`Number` in the example above). Moreover, new values are only stored (and the optional apply method called) if different from the existing values. A tiny but important optimization.

Change Events

qooxdoo supports full-featured event-based programming throughout the framework. So-called *change events* are a good example for this powerful concept.

Each property may optionally behave as an observable. This means it can send out an event at any time the property value changes. Such a change event (an instance of `qx.event.type.Data`) is declared by providing a custom name in the `event` key of the property definition. While you are free to choose any event name you like, the qooxdoo framework tries to consistently use the naming convention `"change + Propertyname"`, e.g. `"changeWidth"` for a change of property `width`. In order to get notified of any value changes, you simply attach an event listener to the object instance containing the property in question.

For example, if you would like the `element` property of a Widget instance `widget` to fire an event named `"changeElement"` any time the value changes. If this happens, you would like to set the DOM element's content:

```
widget.addEventListener("changeElement", function(e) {
  e.getValue().innerHTML = "Hello World";
});
```

The anonymous function acts as an event handler that receives the event object as variable `e`. Calling the predefined method `getValue()` returns the new value of property `element`.

Available Methods

qooxdoo's dynamic properties not only make sure that all properties behave in a consistent way, but also guarantee that the API to access and manipulate properties are identical. The user is only confronted with a single interface, where the method names are easy to understand. Each property creates (at least) the following set of methods:

- `setPropertyName()`: Mutator method ("setter") to set a new property value.
- `getPropertyName()`: Accessor method ("getter") that returns the current value.

Additionally, all properties of boolean type (declared by `check: "Boolean"`) provide the following convenience methods:

- `isPropertyName()`: Identical to `getPropertyName()`.
- `togglePropertyName()`: Toggles between true and false.

Property Groups

Property groups is a layer above the property system explained in the last paragraphs. They make it possible to set multiple values in one step using one `set` call. `qx.ui.core.Widget` supports the property group `padding`. `padding` simply sets the `paddingLeft`, `paddingRight`, `paddingTop` and `paddingBottom` property.

```
widget.setPadding(10, 20, 30, 40);
```

The result is identical to:

```
widget.setPaddingTop(10);
widget.setPaddingRight(20);
widget.setPaddingBottom(30);
widget.setPaddingLeft(40);
```

As you can see the property groups are a nice really convenient feature.

Shorthand support

One more thing. The property group handling also supports some CSS like magic like the shorthand mode for example. This means that you can define only some edges in one call and the others get filled automatically:

```
// four arguments
widget.setPadding(top, right, bottom, left);

// three arguments
widget.setPadding(top, right+left, bottom);

// two arguments
widget.setPadding(top+bottom, right+left);

// one argument
widget.setPadding(top+right+bottom+left);
```

As you can see this can also reduce the code base and make it more userfriendly.

BTW: The values of a property group can also be given an array as first argument e.g. these two lines work identically:

```
// arguments list
widget.setPadding(10, 20, 30, 40);

// first argument as array
widget.setPadding([10, 20, 30, 40]);
```

Note: For more information regarding declaration, usage and internal functionality please see the *the developer documentation*.

3.2.2 Properties in more detail

Note: Please take a look at *Property features summarized* first to get an compact overview of the available features.

Declaration

The following code creates a property `myProperty` and the corresponding functions like `setMyProperty()` and `getMyProperty()`.

```
qx.Class.define(
...
properties : {
  myProperty : { nullable : true }
}
...
```

You should define at least one of the attributes `init`, `nullable` or `inheritable`. Otherwise, the first call to the getter would stop with an exception because the computed value is not (yet) defined.

Note: As an alternative to the `init` key you could set the init value of the property by calling an initializing function `this.initMyProperty(value)` in the constructor. See below for details.

Please also have a look at the *Quick Reference*.

Handling changes of property values

You have multiple possibilities to react on each property change. With *change* the modification of a property is meant, where the old and the new values differ from each other.

As a class developer the easiest solution with the best performance is to define an apply method. As a user of a class (the one who creates instances) it is the best to simply attach an event listener to the instance, if such an corresponding event is provided in the property declaration.

Defining an apply method

To attach an apply method you must add a key `apply` to your configuration which points to a name of a function which needs to be available in your `members` section. As the apply method normally should not be called directly, it is always a good idea to make the method at least protected by prefixing the name with an underscore _.

The return value of the apply method is ignored. The first argument is the actual value, the second one is the former or old value. The last argument is the name of the property which can come very handy if you use one apply mehtod for more than one property. The second and third arguments are optional and may be left out.

Example

```
properties : {
  width : { apply : "_applyWidth" }
},

members :
{
  _applyWidth : function(value, old, name) {
    // do something...
  }
}
```

The applying method is only called when the value has changed.

For a more technical description, take a look at the API documentation of qx.core.Property

Providing an event interface

For the users of a class it is in many cases a nice idea to also support an event to react on property changes. The event is defined using the `event` key where the value is the name of the event which should be fired.

qooxdoo fires a `qx.event.type.Data` which supports the methods `getData()` and `getOldData()` to allow easy access to the new and old property value, respectively.

Note: Events are only useful for public properties. Events for protected and private properties are usually not a good idea.

Example

```
properties : {
  label : { event : "changeLabel" }
}
...
// later in your application code:
obj.addListener("changeLabel", function(e) {
  alert(e.getValue());
});
```

Init values

Init values are supported by all properties. These values are stored separately by the property engine. This way it is possible to fallback to the init value when property values are being reset.

Defining an init value

There are two ways to set an init value of a property.

Init value in declaration The *preferred* way for regular init values is to simply declare them by an `init` key in the property configuration map. You can use this key standalone or in combination with `nullable` and/or `inheritable`.

```
properties : {
  myProperty : { init : "hello" }
}
```

Init value in constructor Alternatively, you could set the init value of the property in the constructor of the class. This is only recommended for cases where a declaration of an init value as explained above is not sufficient.

Using an initializing function `this.initMyProperty(value)` in the constructor would allow you to assign complex non-primitive types (so-called "reference types" like `Array`, `Object`) that should not be shared among instances, but be unique on instance level.

Another scenario would be to use a localizable init value when *internationalizing your application*: Because `this.tr()` cannot be used in the property definition, you may either use the static `qx.locale.Manager.tr()` there instead, or use `this.tr()` in the call of the initializing function in the constructor.

Note: You need to add a `deferredInit:true` to the property configuration to allow for a deferred initialization for reference types as mentioned above.

```
qx.Class.define("qx.MyClass", {
  construct: function() {
    this.initMyProperty([1, 2, 4, 8]);
  },
  properties : {
    myProperty : { deferredInit : true}
  }
};
```

Applying an init value

It is possible to apply the init value using an user-defined apply method. To do this call the init method `this.initMyProperty(value)` somewhere in your constructor - this "change" will than trigger calling the apply method. Of course, this only makes sense in cases where you have at least an `apply` or `event` entry in the property definition.

If you do not use the init method you must be sure that the instances created from the classes are in a consistent state. The getter will return the init value even if not initialized. This may be acceptable in some cases, e.g. for properties without `apply` or `event`. But there are other cases, where the developer needs to be carefully and call the init method because otherwise the getter returns wrong information about the internal state (due to an inconsistency between init and applied value).

Like calling the `this.initMyProperty(value)` method itself, you could call the setter and use the defined init value as parameter. This will call the apply method, not like in the usual cases when setting the same value which is aready set.

```
construct : function()
{
  this.base(arguments);

  this.setColor("black"); // apply will be invoked
  this.setColor("black"); // apply will NOT be invoked
},

properties :
{
  color :
  {
    init : "black",
    apply : "_applyColor"
  }
},

members :
```

```
{
  _applyColor : function(value, old) {
    // do something...
  }
}
```

This example illustrates how the behavior differs from the default behavior of the property system due to the already mentioned inconsistency between init and applied value.

```
construct : function()
{
  this.base(arguments);

  // Initialize color with predefined value
  this.initColor();

  // Initialize store with empty array
  this.initStore([]);
},

properties :
{
  color :
  {
    init : "black",
    apply : "_applyColor"
  },

  store : {
    apply : "_applyStore"
  }
},

members :
{
  _applyColor : function(value, old) {
    // do something...
  },

  _applyStore : function(value, old) {
    // do something...
  }
}
```

In the above example you can see the different usage possibilities regarding properties and their init values. If you do not want to share "reference types" (like `Array`, `Object`) between instances, the init values of these have to be declared in the constructor and not in the property definition.

If an `init` value is given in the property declaration, the init method does not accept any parameters. The init methods always use the predefined init values. In cases where there is no `init` value given in the property declaration, it is possible to call the init method with one parameter, which represents the init value. This may be useful to apply reference types to each instance. Thus they would not be shared between instances.

Note: Please remember that init values are not for incoming user values. Please use `init` only for class defined things, not for user values. Otherwise you torpedo the multi-value idea behind the dynamic properties.

Refining init values

Derived classes can refine the init value of a property defined by their super class. This is however the only modification which is allowed through inheritance. To refine a property just define two keys inside the property (re-)definition: `init` and `refine`. `refine` is a simple boolean flag which must be configured to true.

Normally properties could not be overridden. This is the reason for the `refine` flag . The flag informs the implementation that the developer is aware of the feature and the modification which should be applied.

```
properties : {
  width : { refine : true, init : 100 }
}
```

This will change the default value at definition time. `refine` is a better solution than a simple `set` call inside the constructor because it the initial value is stored in a separate namespace as the user value and so it is possible for the user to fall back to the default value suggested by the developer of a class.

Checking incoming values

You can check incoming values by adding a `check` key to the corresponding property definition. But keep in mind that these checks only apply in the development (source) version of the application. Due to performance optimization, we strip these checks for the build version. If you want a property validation, take a look at the *validation section*.

Predefined types

You can check against one of these predefined types:

- `Boolean`, `String`, `Number`, `Integer`, `Float`, `Double`
- `Object`, `Array`, `Map`
- `Class`, `Mixin`, `Interface`, `Theme`
- `Error`, `RegExp`, `Function`, `Date`, `Node`, `Element`, `Document`, `Window`, `Event`

Due to the fact that JavaScript only supports the `Number` data type, `Float` and `Double` are handled identically to `Number`. Both are still useful, though, as they are supported by the Javadoc-like comments and the API viewer.

```
properties : {
  width : { init : 0, check: "Integer" }
}
```

Possible values

One can define an explicit list of possible values:

```
properties : {
  color: { init : "black", check : [ "red", "blue", "orange" ] }
}
```

Note: Providing a list of possible values only works with primitive types (like strings and numbers), but not with reference types (like objects, functions, etc.).

Instance checks

It is also possible to only allow for instances of a class. This is not an explicit class name check, but rather an `instanceof` check. This means also instances of *any* class derived from the given class will be accepted. The class is defined using a string, thereby to not influencing the load time dependencies of a class.

```
properties : {
  logger : { nullable : true, check : "qx.log.Logger" }
}
```

Interface checks

The incoming value can be checked against an interface, i.e. the value (typically an instance of a class) must implement the given interface. The interface is defined using a string, thereby not influencing the load time dependencies of a class.

```
properties : {
  application : { check : "qx.application.IApplication" }
}
```

Implementing custom checks

Custom checks are possible as well, using a custom function defined inside the property definition. This is useful for all complex checks which could not be solved with the built-in possibilities documented above.

```
properties :
{
  progress :
  {
    init : 0,
    check : function(value) {
      return !isNaN(value) && value >= 0 && value <= 100;
    }
  }
}
```

This example demonstrates how to handle numeric values which only accept a given range of numbers (here 0 .. 100). The possibilities for custom checks are only limited by the developer's imagination. ;-)

Alternative solution As an alternative to the custom check *function*, you may also define a *string* which will directly be incorporated into the setters and used in a very efficient way. The above example could be coded like this:

```
properties :
{
  progress :
  {
    init : 0,
    check : "!isNaN(value) && value >= 0 && value <= 100"
  }
}
```

This is more efficient, particularly for checks involving rather small tests, as it omits the function call that would be needed in the variant above.

Transforming incoming values

You can transform incoming values before they are stored by using the transform key to the corresponding property definition. The transform method occurs before the check and apply functions and can also throw an error if the value passed to it is invalid. This method is useful if you wish accept different formats or value types for a property.

Example

Here we define both a check and transform method for the width property. Though the check method requires that the property be a integer, we can use the transform method to accept a string and transform it into an integer. Note that we can still rely on the check method to catch any other incorrect values, such as if the user mistakenly assigned a Widget to the property.

```
properties :
{
   width :
   {
      init : 0,
      transform: "_transformWidth",
      check: "Integer"
   }
},

members :
{
   _transformWidth : function(value)
   {
      if ( qx.lang.Type.isString(value) )
      {
          value = parseInt(value, 10);
      }

      return value;
   }
}
```

Validation of incoming values

Validation of a property can prevent the property from being set if it is not valid. In that case, a validation error should be thrown by the validator function. Otherwise, the validator can just do nothing.

Using a predefined validator

If you use predefined validators, they will throw a validation error for you. You can find a set of predefined validators in qx.util.Validate. The following example shows the usage of a range validator.

```
properties : {
  application : { validate : qx.util.Validate.range(0, 100) }
}
```

Using a custom validator

If the predefined validators are not enough for you validation, you can specify your own validator.

```
properties : {
  application : { validate : function(value) {
      if (value > 10) {
        throw new qx.core.ValidationError(
          "Validation Error: ", value + " is greater than 10."
        );
      }
    }
  }
}
```

Validation method as member

You can define a validation method as a member of the class containing the property. If you have such a member validator, you can just specify the method name as a sting

```
properties : {
  application : { validate : "_validateApplication" }
}
```

Enabling theme support

The property system supports *multiple values per property* as explained in the paragraph about the init values. The theme value is another possible value that can be stored in a property. It has a lower priority than the user value and a higher priority than the init value. The `setThemed` and `resetThemed` methods are part of qooxdoo's theme layer and should not be invoked by the user directly.

```
setter                                       value                    resetter

setProperty(value)               ^           user           |         resetProperty()
                                 |                          |
setThemedProperty(value)    Priority         theme      Fallback      resetThemedProperty()
                                 |                          |
initProperty([value])            |           init           v         n.a.
```

To enable theme support it is sufficient to add a `themeable` key to the property definition and set its value to `true`.

```
properties : {
  width : { themeable : true, init : 100, check : "Number" }
}
```

Note: `themeable` should only be enabled for truely *theme-relevant* properties like color and decorator, but not for *functional* properties like enabled, tabIndex, etc.

Working with inheritance

Another great feature of the new property system is inheritance. This is primarily meant for widgets, but should be usable in independent parent-children architectures, too.

Inheritance quickly becomes nothing short of vital for the property system, if you consider that it can reduce redundancy dramatically. It is advantageous both in terms of coding size and storage space, because a value only needs to be declared once for multiple objects inside an hierarchy. Beyond declaring such an inheritable property once, only intended exceptions to the inherited values need to be given to locally override those values.

The inheritance as supported by qooxdoo's properties is comparable to the inheritance known from CSS. This means, for example, that all otherwise undefined values of inheritable properties automatically fall back to the corresponding parent's value.

Each property may also have an explicit user value of string `"inherit"`. The inherited value, which is normally only used as a fallback value, can thus be emphasized by setting `"inherit"` explicitly. The user may set a property to `"inherit"` in order to enforce lookup by inheritance, and thereby ignoring init and appearance values.

To mark a property as inheritable simply add the key `inheritable` and set it to `true`:

```
properties : {
  color : { inheritable : true, nullable : true }
}
```

Optionally, you can configure an init value of `inherit`. This is especially a good idea if the property should not be nullable:

```
properties : {
  color : { inheritable : true, init: "inherit" }
}
```

Inheritable CSS properties

To give you an idea for what kind of custom properties inheritance is particularly useful, the following list of prominent CSS properties which support inheritance may be a good orientation:

- `color`
- `cursor`
- `font`, `font-family`, ...
- `line-height`
- `list-style`
- `text-align`

Note: This list of CSS properties is only meant for orientation and does not reflect any of qooxdoo widget properties.

Internal methods

The property documentation in the user manual explains the public, non-internal methods for each property. However, there are some more, which are not meant for public use:

- `this.resetProperty(value)` : For properties which are inheritable. Used by the inheritance system to transfer values from parent to child widgets.
- `this.setThemedProperty(value)` : For properties with `appearance` enabled. Used to store a separate value for the appearance of this property. Used by the appearance layer.
- `this.resetThemedProperty(value)` : For properties with `appearance` enabled. Used to reset the separately stored appearance value of this property. Used by the appearance layer.

Defining property groups

Property groups is a convenient feature as it automatically generates setters and resetters (but no getters) for a group of properties. A definition of such a group reads:

```
properties : {
  location : { group : [ "left", "top" ] }
}
```

As you can see, property groups are defined in the same map as "regular" properties. From a user perspective the API with setters and resetters is equivalent to the API of regular properties:

```
obj.setLocation( 50, 100);

// instead of
// obj.setLeft(50);
// obj.setTop(100);
```

Shorthand support

Additionaly, you may also provide a mode which modifies the incoming data before calling the setter of each group members. Currently, the only available modifier is `shorthand`, which emulates the well-known CSS shorthand support for qooxdoo properties. For more information regarding this feature, please have a look at the *user manual*. The definition of such a property group reads:

```
properties :
{
  padding :
  {
    group : [ "paddingTop", "paddingRight", "paddingBottom", "paddingLeft" ],
    mode : "shorthand"
  }
}
```

For example, this would allow to set the property in the following way:

```
obj.setPadding( 10, 20 );

// instead of
// obj.setPaddingTop(10);
// obj.setPaddingRight(20);
// obj.setPaddingBottom(10);
// obj.setPaddingLeft(20);
}
```

```
    .. _pages/defining_properties#when_to_use_properties:
```

When to use properties?

Since properties in qooxdoo support advanced features like validation, events and so on, they might not be quite as lean and fast as an ordinarily coded property that only supports a setter and getter. If you do not need these advanced features or the variable you want to store is *extremely* time critical, it might be better not to use qooxdoo's dynamic properties in those cases. You might instead want to create your own setters and getters (if needed) and store the value just as a hidden private variable (e.g. `__varName`) inside your object.

3.2.3 Initialization Behavior

This document summarizes some thoughts about the behavior of the initialization of properties.

The Problem

Imagine a class containing a property named `a` with an init value, like the following:

```
qx.Class.define("A", {
        extend : qx.core.Object,
        properties : {
          a : {
            init : "b",
            event : "changeA"
          }
        }
      });
```

As you can see, the property `a` has an init value, `b`. Now, if you access `a` with its getter, you get the init value in return:

```
var a = new A();
a.getA();  // returns "b"
```

If you now set something different than the initial value, you get a change event, because the content of the property changed.

```
a.setA("x");  // changeA fired
```

As far, everything behaves as desired. But if set the init value instead of a new value, the change event will be also fired. The following code shows the problem:

```
var a = new A();
a.setA(a.getA()); // changeA fired (first set)
a.setA(a.getA()); // changeA NOT fired (every other set)
```

Why not just change this behaviour?

It's always hard to change a behavior like that because there is no deprecation strategy for it. If we change it, it is changed and every line of code relying on that behavior will fail. Even worse, the only thing we could use as a check for the wrong used behavior is to search for all properties having an init value and either an apply function or an event. Now you have to check if one of these properties could be set with the init value, before any other value has been set. If it is possible that the init value is set as first value, check if the attached apply is required to run or any listener registered to the change event of that property. A good example in the framework where we rely on the behavior is the Spinner:

```
// ...
  construct : function(min, value, max) {
// ...
    if (value !== undefined) {
      this.setValue(value);
    } else {
      this.initValue();
    }
// ...
    _applyValue: function(value, old)
// ...
```

```
            this._updateButtons();
// ...
```

The example shows the constructor and the apply of the value property. The problem begins in this case with the constructor parameter named `value`, which is optional. So we have three cases to consider.

1. The value argument is undefined: The initValue method is called, which invokes the apply function for the property with the init value as value.
2. A value is given different as the init value: So the value is not undefined and the setter for the value property will be called, which invokes the apply function.
3. A value is given and its exactly the init value: In this case, the setter will be called with the init value. The apply method is called and invokes the `_updateButtons` method. This method checks the given value and enables / disabled the buttons for increasing / decreasing the spinner. So it is necessary that the apply method is at least called once that the buttons have the proper states.

The problem with a possible change of this behavior is obvious. In the third case, the apply method is not called and the buttons enabled states could be wrong without throwing an error. And they are only wrong, if the value is exactly the init value and one of the minimum or maxiumum values is the same. Because only in that scenario, one of the buttons need to be disabled.

When can it be changed?

Currently we don't plan to change it because it can have some hard to track side effects as seen in the example above and we don't have any deprecation strategy. Maybe it can be change on a major version like 2.0 but currently there are no plans to do so.

3.2.4 Property features summarized

Note: The chapter gives you an compact but extensive overview of the features offered by qooxdoo's property system. Please refer to *Properties in more detail* for an explanation of how to define and use properties.

Value checks

- Built-in types for most common things
- Runtime checks (development version only)
- Instance checks by simply define the classname of the class to check for (always use an instanceof operation - a real classname is not available anymore)
- Custom check method by simply attaching a function to the declaration
- Custom check defined by a string which will be compiled into the resulting setters (faster than the above variant)
- Define multiple possible (primitive) values using an array

Validation

- Validation in both development and build version
- Predefined validators for default validation
- Throws a special validation error

Advanced value handling

- Multi value support. Support to store different values for init, inheritance, style and user including a automatic fallback mechanism between them.
- Inheritance support. Inheritance of properties defined by a parent widget e.g. inherit enabled from a groupbox to all form elements. Uses the inheritance if the computed value would be `undefined` or explicitly set to `"inherit"`. The getter simply returns `"inherit"` for inheritable properties which are otherwise unset.
- Blocks unintentionally `undefined` values in all setters with an exception. To reset a value one must use the `reset` or `unstyle` method which are available too.
- Overriding of a value by setting a property explicitly to `null`
- Properties must be explicitly configured as `"nullable"` (like in .Net). The default is `false` which means that incoming `null` values will result in an exception.
- Accessing nullable properties with `undefined` values will result in a normalization to `null`.

Convenience

- Convenient toggle method for boolean properties

Notification

- Support for a custom apply rountine
- Event firing with a custom named event

Initialization

qooxdoo automatically correctly initializes properties. This is true for both, properties which have defined an `init` value and also for the other properties which are `nullable`. This means that after you have created an instance the properties correctly reflect the applied value. Default values assigned by `init` also execute the configured `apply` methods and dispatch configured events to inform already added listeners.

Initialization Behavior

Performance

Automatic optimization of all setters to the optimal highly-tuned result code. Impressive tailor made high performance setters are the result.

Please note that after the definition point of a property the setters are not yet available. Wrappers for them will be created with the first instance and the final code will be generated with the first use of such a setter. This first use will also automatically unwrap the property setter to directly use the generated one.

Memory managment

Automatic memory management. This means all so-configured properties which contain complex data objects get automatically disposed with the object disposal. The affected built-in types are already auto-configured this way. Also all properties which need an instance of a class, defined by using a classname as `check` are automatically handled.

Note: Note that this does not actually call dispose() on the object but just removes the property value etc i.e. dereferences the object. You still need to call dispose() if necessary.

For all other properties which contain complex data the developer must add a `dispose` key with a value of `true` to the property declaration. For example if there is no `check` defined or the `check` definition points to a function.

Note: This is not needed for primitive types like strings and numbers.

3.3 Settings and Variants

3.3.1 Settings

One of the major problems of JavaScript frameworks is that you, as the user of such a framework, cannot easily control one of the initial settings. For example the framework may have defaults which can only be changed after the framework is loaded, but not before. Most of the time this restriction is not problematic. Most applications are only interested in settings once their main routine gets processed. But there are exceptions when things must be configured at or before load time.

What are settings?

This is where qooxdoo's settings come in. They are directly built into the core of qooxdoo. This means that many intial settings are easily controllable using a simple hash map structure or generator settings.

For example you can control the following things in qooxdoo:

- All type of themes (colors, icons, widgets, appearance)
- Default log level and appender
- Resource-URLs of standard qooxdoo icons and widgets images
- Timeout of the image preloader
- The init component (graphical or non-graphical)
- Different debugging options for json, remote io, etc.

This list shows you some of your possibilities.

When to use settings (and when not)

Generally settings are not meant to replace the properties which exist in qooxdoo. **Settings are immutable**. There is only a default value, defined by the class which declares the setting, and a optional user value which overrides this default value. The only possiblity to change this user value is to use the generator option or to define a global map `qxsettings` before including the qooxdoo JavaScript file(s). It is not possible to modify the value later.

As a class developer it is quite important to not use settings for everything. The intention is to reduce the number of settings to a minimum. They are only meant to control loadtime relevant stuff. Other things should be resolved with other available technologies like properties. Properties have a much greater set of features and possibilities.

Defining new settings

New settings can be defined in this way:

```
qx.core.Setting.define("ns.key", "value");
```

In contrast to *Variants* there is no array of available values. Settings are not limited in this way. You can even store any JS-Type in a setting. But normally it is better to just use primitive types like Boolean, String and Number values.

The qooxdoo class definition allows you to integrate settings directly in the defining map:

```
qx.Class.define("myapp.ClassA",
{
  [...]

  settings : {
    "myapp.key" : value
  }
});
```

The key should always contain a namespace. This protects the application developer from creating conflicting settings with the framework and maybe other qooxdoo-based libraries. The namespace could be something short. All qooxdoo settings use "qx", like the toplevel namespace. If you have a "myapp.Application" you may want to prefix all your settings with "myapp" (but deeper nested namespaces are also possible).

Even if settings (and variants) are defined in the class where they are used, they are stored in a more "global" manner and may be accessed from everywhere.

Note: Important: You must be sure that the class defining –which is using a setting– is loaded before the first access to this setting. Also you must not redefine settings.

Selecting settings

You can select a new value for a settings using the generator or a global map defined before loading qooxdoo.

Using the generator

Use the config.json *settings* config key:

```
{
  ...
  "settings" : {
    "myapp.key" : "value",
    ...
  }
}
```

Using a map

Use a map named `qxsettings` which is globally defined in the loading HTML page

```
window.qxsettings = {
  "myapp.key" : value,
  ...
}
```

Note: Because of the namespace-like dot you must be sure to put the whole hash map key into quotes.

Predefined Settings

These settings are known in the qooxdoo framework:

Setting	Recognized Values	Default
qx.allowUrlSettings	true/false	false
qx.allowUrlVariants	true/false	false
qx.application	<string>	<undefined>
qx.bom.htmlarea.HtmlArea.debug	"on"/"off"	"off"
qx.disposerDebugLevel	0, 1, ...	0
qx.globalErrorHandling	"on"/"off"	"on"
qx.ioRemoteDebug	true/false	false
qx.ioRemoteDebugData	true/false	false
qx.jsonEncodeUndefined	true/false	true
qx.jsonDebugging	true/false	false
qx.nativeScrollBars	true/false	false
qx.propertyDebugLevel	0, 1, ...	0
qx.tableResizeDebug	true/false	false

3.3.2 Variants

Variants enable the selection and removal (in the build version) of code. A variant is a named value from a finite collection from which exactly one is set at load time of the framework. The static class `qx.core.Setting` can be used to query the value of a setting.

Depending on the current value of a variant a specific code path can be chosen using the `qx.core.Setting.select` method.

You can set variants in the generator configuration. The generator will then set this variant and for the build version will remove all code paths which are not selected by the current variant value.

Variants are used to implement browser optimized builds and to remove debugging code from the build version. It is very similar to conditional compilation in C/C++. In the source version of the application, calls to `select` are like if/case switches.

Browser optimized builds

qooxdoo tries to hide browser incompatibilities from the application developer. But to provide browser independent functionality it is sometimes necessary to use different code on different browsers. Low level code like the key handler often has an individual implementation for each supported browser (and uses browser variants to achieve this).

The generator selects for browser optimized builds only the code which is needed for one specific browser and removes the unused code. For each supported browser engine an optimized build is generated and on load time the appropriate build is loaded. As a fall back there can be an unoptimized build.

Code like this was very common in older versions of qooxdoo:

```
if (qx.core.Client.getInstance().isMshtml()) {
  // some Internet Explorer specific code
} else if(qx.core.Client.getInstance().isOpera()){
  // Opera specific code
} else {
  // common code for all other browsers
}
```

Using `Variants` the same code looks like this:

```
if (qx.core.Variant.isSet("qx.client", "mshtml")) {
  // some Internet Explorer specific code
} else if(qx.core.Variant.isSet("qx.client", "opera")){
  // Opera specific code
} else {
  // common code for all other browsers
}
```

The variant `qx.client` is always set to the current browser, so this code works exactly like the first version. What is new is that the generator knows about variants and is able to optimize the build for one value of a variant and remove the unused code for all other values of the variant.

Browser optimization is enabled by default in skeleton based applications.

Removal of debugging code

Often one wants to add additional checks and assertions to the code but don't want the build to suffer from these checks. This can be solved elegantly by using variants too. The variant `qx.debug` with the allowed values `on` and `off` can be used to add debugging code which is only active in the source version and removed from the build version.

Example:

```
function foo(a, b) {
  if (qx.core.Variant.isSet("qx.debug", "on")) {
    if ( (arguments.length != 2) || (typeof a != "string") ) {
      throw new Error("Bad arguments!");
  }
}
```

This check is now only enabled in the source version. By default `qx.debug` is set to `off` in build versions.

Details

Variants are used to select certain code paths. Each variant has a name and exactly one value from a limited list of allowed values. The variant names have a namespace prefix to avoid name conflicts. The value of a variant is immutable and once set cannot be altered in the JavaScript code.

Definition of a Variant

The method `qx.core.Variant.define` is used to define a variant. This is how `qx.debug` is defined:

```
qx.core.Variant.define("qx.debug", [ "on", "off" ], "on");
```

The first parameter is the name of the variant, the second is a string array of all allowed values and the third the default value. The default is taken if the variant is not set otherwise.

Using Variants

Variants can be used in two ways. They can be used to select code using `if` statements or to select whole functions.

select If the whole definition of a function should be selected the `select` method can be used as follows:

```
var f = qx.core.Variant.select("qx.client", {
  "gecko": function() { ... },
  "mshtml|opera": function() { ... },
  "default": function() { ... }
});
```

Depending on the value of the `qx.client` variant the corresponding function is selected. The first case is selected if the variant is *gecko*, the second is selected if the variant is *mshtml* or *opera* and the third function is the default case. It is selected if none of the other keys match the variant.

isSet This method is used to check whether a variant is set to a given value. The first parameter is the name of the variant and the second parameter is the value to check for. Several values can be "or"-combined by separating them with the "|" character. A value of *mshtml|opera* would for example check whether the variant is set to "mshtml" or "opera".

To enable the generator to optimize this selection, both parameters must be string literals.

This method is meant to be used in *if* statements to select code paths. If the condition of an *if* statement is only this method, the generator is able to optimize the statement.

Example:

```
if (qx.core.Variant.isSet("qx.client", "mshtml")) {
  // some Internet Explorer specific code
} else if(qx.core.Variant.isSet("qx.client", "opera")){
  // Opera specific code
} else {
  // common code for all other browsers
}
```

Setting the Value of a Variant

There are three ways to set a variant:

- Setting the value in the global variable `qxvariants` before qooxdoo is loaded.
- Set the variant in the generator configuration, using the `variants` config key.
- Set the variant in JS class code, using `qx.core.Variant.define`.

For the first approach just define a global map named `qxvariants`. This is how it could look in your application application:

```
<script language="JavaScript" type="text/javascript">
qxvariants = {
    "custom.variant": "off"
}
</script>
<script language="JavaScript" type="text/javascript" src="script/qooxdoo_application.js"></script>
```

Predefined Variants

Here is a list of variants currently predefined in qooxdoo:

Variant	Possible Values	Default
"qx.aspects"	["on", "off"]	"off"
"qx.client"	["gecko", "mshtml", "opera", "webkit"]	qx.bom.client.Engine.NAME
"qx.debug"	["on", "off"]	"on"
"qx.dynlocale"	["on", "off"]	"off"

3.4 Data Binding

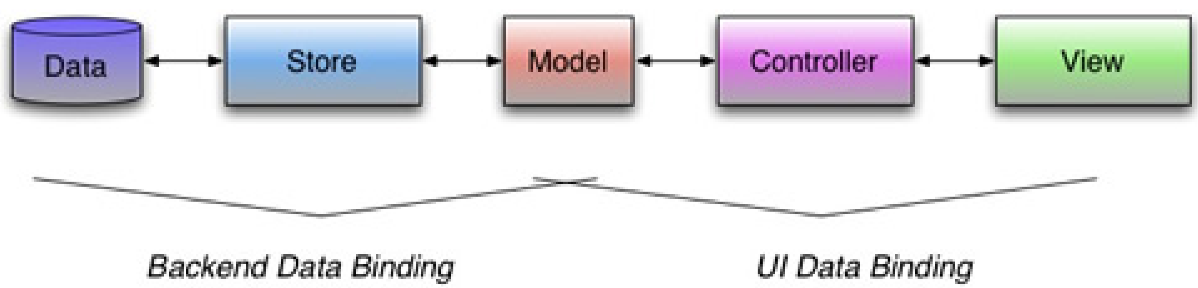
3.4.1 Data Binding

Data Binding

Introduction

Data binding is a functionality that allows to connect data from a source to a target. The entire topic can be divided into a low-level part, called "single value binding", and some higher-level concepts involving stores and controllers.

The main idea The main idea of qooxdoo's data binding component is best summarized by the following diagram.



As you can see data binding includes five major components, which will be described in more detail in the following sections.

Data The data part is where the raw data is stored and can be retrieved from. This can be a plain local file, a regular web server or even a web service. There are all sources of data possible depending on the implementation of the actual store.

Store The store component is responsible for fetching the data from its source and for including it into a data model of an application. For more info about the available store components see the *stores section* below.

Model The model is the centerpiece of data binding. It holds the data and acts as an integration point for the store and for the controller. Almost all models are plain qooxdoo classes holding the data in properties, which are configured to fire events on every change. Since native JavaScript arrays do not fire events when items are changed, a complementary array is added for data binding purposes. It is available with most of the native array API as `qx.data.Array`. But there is no need to manually write own model classes for every data source you want to work with. The stores provide a smart way to automatically create these classes during runtime. Take a look at the *stores section* for details.

Controller The main task of the controller components is to connect the data in the model to the view components. Details are available in the *controller section*. The base layer of all controllers, the *Single Value Binding* is explained later.

View The views for data binding can be almost any widget out of qooxdoo's rich set of widgets, depending on the type of controller. qooxdoo's data binding is not limited to some predefined data bound widgets. Please note that one of the most prominent data centric widgets, the virtual Table, currently still has its own model based layer and is not covered by the new data binding layer. The new infrastructure for virtual widgets is expected to nicely integrate the upcoming data binding layer, though.

Demos, API and CheatSheet

You should now have a basic idea of qooxdoo's data binding, so to see it in action, take a look at the online demos and the API reference. If you want to start programming, maybe the `CheatSheet` can help you during your programming.

Single Value Binding

The purpose of single value binding is to connect one property to another by tying them together. The connection is always in one direction only. If the reverse direction is needed, another binding needs to be created. The binding will be achieved by an event handler which assigns the data given by the event to the target property. Therefore it is necessary for the source event to fire a change event or some other kind of data event. The single value binding is mostly a basis for the higher concepts of the data binding.

Binding a single property to another property

The simplest form of single value binding is to bind one property to another. Technically the source property needs to fire a change event. Without that no binding is possible. But if this requirement is met, the binding itself is quite simple. You can see this in the following code snippet, which binds two properties of the label content together:

```
var label1 = new qx.ui.basic.Label();
var label2 = new qx.ui.basic.Label();

label1.bind("content", label2, "content");
```

`label1` is the source object to bind, with the following three arguments to that call:

1. The name of the property which should be the source of the binding.
2. The target object which has the target property.
3. The name of the property as the endpoint of the binding.

With that code every change of the content property of `label1` will automatically synchronize the content property of `label2`.

Binding a data event to property

In some cases in the framework, there is only a change event and no property. For that case, you can bind a data event to a property. One common case is the `TextField` widget, which does not have a property containing the content of the `TextField`. Therefor you can use the input event and bind that to a target property as you can see in the example snippet. The API is almost the same as in the property binding case.

```
var textField = new qx.ui.form.TextField();
var label = new qx.ui.basic.Label();

textField.bind("input", label, "content");
```

As you can see, the same method has been used. The difference is, that the first argument is a data event name and not a property name.

Bind a property chain to another property

A more advanced feature of the single value binding is to bind a hierarchy of properties to a target property. To understand what that means take a look at the following code. For using that code a qooxdoo class is needed which is named `Node` and does have a `child` and a `name` property, both firing change events.

```
// create the object hierarchy
var a = new Node("a");      // set the name to „a“
var b = new Node("b");      // set the name to „b“
a.setChild(b);

// bind the property to a labels content
a.bind("child.name", label, "content");
```

Now every change of the `name` of `b` will change the labels content. But also a change of the `child` property of `a` to another Node with another name will change the content of the label to the new name. With that mechanism a even deeper binding in a hierarchy is possible. Just separate every property with a dot. But always keep in mind that every property needs to fire a change event to work with the property binding.

Bind an array to a property

The next step in binding would be the ability to bind a value of an array. That's possible but the array needs to be a special data array because the binding component needs to know when the array changes one of its values. Such an array is the `qx.data.Array` class. It wraps the native array and adds the change event to every change in the array. The following code example shows what a binding of an array could look like. As a precondition there is an object `a` having a property of the `qx.data.Array` type and that array containing strings.

```
// bind the first array element to a label's content
a.bind("array[0]", labelFirst, "content");

// bind the last array element to a label's content
a.bind("array[last]", labelFirst, "content");
```

You can use any numeric value in the brackets or the string value `last` which maps to `length - 1`. That way you can easily map the top of a stack to something else. For binding of an array the same method will be used as for the binding of chains. So there is also the possibility to combine these two things and use arrays in such property chains.

Options: Conversion and Validation

The method for binding introduced so far has the same set of arguments. The first three arguments are mostly the same. There

- **converter**: A own converter which is a function with one argument returning the converted value.
- **onSetOk**: A key in the options map under which you can add a method. This method will be called on a validation case if the validation was successful.
- **onSetFail**: The counterpart to onSetOk which will be called if the validation fails.

In addition there is a built in default conversion which takes care of the default conversion cases automatically. Default cases are, for example, string to number conversion. To get that working it is necessary to know the desired target type. This information is taken from the check key in the property definition of the target property.

Managing bindings

If you want to manage the bindings, there are some ways to get that. First aspect of managing is showing the existing bindings. You can find all these function on the static `qx.data.SingleValueBinding` class or parts of it on every object.

- **getAllBindingsForObject** is a function which is in the data binding class and returns all bindings for the given object. The object needs to be the source object.
- **getAllBindings** returns all bindings in a special map for all objects.

Another way of managing is removing. There are three ways to remove bindings.

- **removeBindingFromObject** removes the given binding from the given source object. As an id you should use exactly the id returned during the creation of the binding.
- **removeAllBindingsForObject** removes all binding from the source object. After that, the object is not synchronized anymore.
- **removeAllBindings** removes all single value bindings in the whole application. Be careful to use that function. Perhaps other parts of the application use the bindings and also that will be removed!

Debugging

Working with bindings is in most cases some magic and it just works. But the worse part of that magic is, if it does not work. For that the data binding component offers two methods for debugging on the static `qx.data.SingleValueBinding` class.

- **showBindingInLog** shows the given binding in the qooxdoo logger as a string. The result could look something like this: *Binding from 'qx.ui.form.TextField[1t]' (name) to the object 'qx.ui.form.TextField[1y]' (name).* That shows the source object and property and the target object and property.
- **showAllBindingsInLog** shows all bindings in the way the first method shows the bindings.

Tech notes

For everyone who is interested on how that whole thing works, here are some small notes on the inside of the data binding. Every binding function maps to the event binding function. This is where the heart of the data binding lies. In that function a listener will be added to the source object listening to the change event. The key part of the listener is the following code part.

```
targetObject["set" + qx.lang.String.firstUp(targetProperty)](data);
```

In that line the listener sets the data given by the data event to the target property.

Controller

The general idea of controllers is connecting a view component to a set of data stored in a model. The kind of controller you need depends on the view component. Currently there are four types of controller available:

- Object Controller
- List Controller
- Tree Controller
- Form Controller

You may miss the table controller. The currently available table will not be changed and therefore will not implement data binding features. The new virtual table, which is currently under development, will be considered for data binding.

In the following section, the selection will be discussed because it's a common feature of the list and tree controller. The delegation mechanism is another common feature of those two controllers and will also be described. After that, each of the available controllers will be discussed in detail.

Selection

Usually the selection of view components like the tree or the list handle their selection with tree folder or list items. As a user of data binding, you don't want to convert the view widgets to the model widgets. Therefore, the controller does that mapping for you. There is a selection array available on the controller containing the currently selected model items. When using the selection of the controller, there is no need to deal with view widgets like ListItems. It is also possible to change the array in place and add / remove something from the selection. As it is a data array, you can use all methods defined by that array to manipulate the selection of the corresponding controller.

Delegate

The list and tree controller are responsible for creating and binding the child widgets of the views. But what if you want to use something different in the list or bind not just the label and the icon. For that purpose, the delegation offers the possibility to enhance the controller code without having to subclass it.

In total, there are three methods which relate to the topic of creating and binding the child view widgets.

configureItem The `configureItem` function is the function which you can use if you just want to modify the created default widgets. This gives you the opportunity to set the labels to rich for example or modify anything else in the child widget. But this is not the place where you want to change / add the binding behavior.

bindItem That place is the `bindItem` method. But you don't want to use the single value binding all on your own and bind the stuff. Therefore, the controller offers you a method called `bindProperty`, which takes the source path to the data, the target property name and the options for the single value binding. The other two parameters will just mapped through. But keep in mind that if you use this function, the default binding of the label and the icon is gone and the properties used for those bindings do not work anymore. If you still want to have the default binding, use the `bindDefaultProperties` method and pass the two given parameters through. But keep in mind that the bindings set up with these two methods are unidirectional, from the model to the view. If you want to have b binding from the view to the model, use the `bindPropertyReverse` which takes the same arguments as the `bindProperty` method.

createItem The last method named `createItem` gives the user the chance to add something different as child widgets to the view. In that method you just create the widget you want to see in the view and return the new item. But keep in mind that the default bindings may not work on those widgets and the code will fail. So it is always a good idea to also define its own bindings with the `bindItem` method.

The following code shows how such a delegate could look like.

```
var delegate = {
  configureItem : function(item) {
    item.setPadding(3);
  },
  createItem : function() {
    return new qx.ui.form.CheckBox();
  },
```

```
  bindItem : function(controller, item, id) {
    controller.bindProperty("name", "label", null, item, id);
    controller.bindProperty("online", "checked", null, item, id);
  }
};
```

The delegate defines, that `CheckBox``es should be used as child view items. As the ``CheckBox``es don't have an icon, the ``bindItem` function needs to re-specify the bindings. It binds the name and the online property of the model to the label and checked property of the `CheckBox`.

Object Controller

The most simple and lightweight controller is the object controller. It connects a model object with one or more views. The data in the model can be anything a property can hold, i.e. a primitive data type like String or Number, or a reference type like a map. With that you can for instance bind views like textfields, sliders and other widgets visualizing primitive JavaScript types. But you can not only use views as targets. A target can be anything that has a property with the proper type. Take a look at the following code example to see the object controller in action:

```
// create two sliders
var slider1 = new qx.ui.form.Slider();
var slider2 = new qx.ui.form.Slider();
// create a controller and use the first slider as a model
var controller = new qx.data.controller.Object(slider1);
// add the second slider as a target
controller.addTarget(slider2, "value", "value");
```

This code snippet ensures that every value set by slider1 will automatically be set as value of slider two. As you can see, the object controller only wraps the fundamental single-value binding, trying to make its usage a little bit easier.

List Controller

A list controller could - as the name suggests - be used for list-like widgets. The supported list-like widgets in qooxdoo are List, SelectBox and ComboBox, all in the qx.ui.form package. The controller expects a data array as a data model, that contains the model objects. These objects are displayed in the list and can either have some primitive type or be real qooxdoo objects. The following code snippet shows how to bind an array of strings to a list widget:

```
// create the model
var model = new qx.data.Array(["a", "b", "c", "d", "e"]);
// create a list widget
var list = new qx.ui.form.List();
// create the controller
var listController = new qx.data.controller.List(model, list);
```

Now every change in the model array will invoke a change in the list widget.

As a unique feature of the list controller a filtering method is included. You can assign a filter function to the controller and the results will be filtered using your given function.

Tree Controller

Of course, also the tree does have its own controller. With that controller the Tree widget can automatically be filled with data from qooxdoo objects containing the data. As model nodes for the tree, only qooxdoo widgets are allowed containing at least two properties, one for holding its own children in a data array and a second one holding the name of the node which should be showed as the label of the tree folder widgets. Imagine that a model class called Node is available containing the two already mentioned properties called ch for the children and n for the name. The following code will bind a data model containing Node objects to a tree widget:

```
// create the model
var rootNode = new qx.Node();
rootNode.setN("root");
var childNode = new qx.Node();
childNode.setN("child");
rootNode.getCh().push(childNode);
// create the tree view
var tree = new qx.ui.tree.Tree();
// create the controller
var treeController = new qx.data.controller.Tree(rootNode, tree, "ch", "n");
```

After that code snippet, every change in the name or of the children will be automatically mapped into the tree view. Selecting one of the tree folders will put the corresponding Node object into the selection array of the controller.

Form Controller

Also forms do have a special controller. The form controller uses a `qx.ui.form.Form` as target and a *Object controller* for the bidirectional bindings. The usage equals to the usage of all other controllers. The main properties of it are the model and target property. Given both, the controller connects the model and the target. An additional feature of the form controller is the possibility to create the model for a given form. See the following code to get an idea of using it.

```
// a form is available as 'form'
// create the controller
var formController = new qx.data.controller.Form(null, form);
// create the model
var model = formController.createModel();
```

If you nee additional information on forms, see *form handling documentation*. After executing this code, the controller and the model variable do have the model available and therefore, the controller can set up the bindings.

Combining Controller

As a more advanced example we connect the selection of a tree to a list. Therefore we extend the code sample of the tree controller section.

```
// create a list widget
var list = new qx.ui.form.List();
// create the controller
var listController = new qx.data.controller.List(null, list, "n");
// bind the selection of the tree to the list
treeController.bind("selection", listController, "model");
```

The example shows how the controller can work pretty well together with the single value binding. The trick is not to set the model of the list controller at creation time. The model will be set by the single value binding from the tree controllers selection. This works because the selection will be provided as data array.

Stores

The main purpose of the store components is to load data from a source and convert that data into a model. The task of loading data and converting the data into a model has been split up. The store itself takes care of loading the data but delegates the creation of model classes and instances to a marshaler.

The only marshaler currently available is for JSON data and therefore, the only data store available is a JSON store. Both will be described in detail in the following sections.

JSON Marshaler

NOTE: This class should only be used if you want to write your own data store for your own data types or request.

The marshaler takes care of converting JavaScript Objects into qooxdoo classes and instances. You can initiate each of the two jobs with a method.

toClass This method converts a given JavaScript object into model classes. Every class will be stored and available in the `qx.data.model` namespace. The name of the class will be generated automatically depending on the data which should be stored in it. As an optional parameter you can enable the inclusion of bubbling events for every change of a property. If a model class is already created for the given data object, no new class will be created.

toModel The method requires that the classes for the models are available. So be sure to call the `toClass` method before calling this method. The main purpose of this method is to create instances of the created model classes and return the model corresponding to the given data object.

createModel (static) This method is static and can be used to invoke both methods at once. By that, you can create models for a given JavaScript objects with one line of code:

```
var model = qx.data.marshal.Json.createModel({a: {b: {c: "test"}}});
```

JSON Store

The JSON store takes an URL, fetches the given data from that URL and converts the data using the JSON marshaler to qooxdoo model instances, which will be available in the model property after loading. The state of the loading process is mapped to a state property. For the loading of the data, a `qx.io.remote.Request` will be used in the store.

The following code shows how to use the JSON data store.

```
var url = "json/data.json";
var store = new qx.data.store.Json(url);
```

After setting the URL during the creation process, the loading will begin immediately. As soon as the data is loaded and converted, you can access the model with the following code.

```
store.getModel();
```

JSONP Store

The JSONP store is based on the *JSON store* but uses a script tag for loading the data. Therefore, a parameter name for the callback and an URL must be specified.

The following code shows how to use the JSONP data store.

```
var url = "json/data.json";
var store = new qx.data.store.Jsonp(url, null, "CallbackParamName");
```

After setting the URL and the callback parameter name during the creation process, the loading will begin immediately.

YQL Store

YQL is the Yahoo! Query Language. Yahoo! describes it as *"[...] an expressive SQL-like language that lets you query, filter, and join data across Web services."* Based on the *JSONP store*, qooxdoo offers a YQL store, where you can specify the YQL queries and qooxdoo handles the rest.

The following code demonstrates how to fetch some twitter messages.

```
var query = "select * from twitter.user.timeline where id='wittemann'";
var store = new qx.data.store.YQL(query);
```

Combining with controllers

As described in the section above, you can access the model in the property after loading. The best solution is to use the model with a controller and then bind the the model properties with single value binding together. The code for this could look something like this.

```
store.bind("model", controller, "model");
```

Using the single value binding, the binding handles all the stuff related with the loading of the model data. That means that the data will be available in the controller as soon as its available in the store.

How to get my own code into the model?

What if you want to to bring your own code to the generated model classes or if you even want to use your own model classes? Thats possible by adding and implementing a delegate to the data store. You can either

- Add your code by supporting a superclass for the created model classes.
- Add your code as a mixin to the created model classes.
- Use your own class instead of the created model classes.

Take a look at the API-Documentation of the `` `qx.data.store.IStoreDelegate <http://demo.qooxdoo.org/1.3.1/apiviewer/index.html#qx.data.store.IStoreDelegate>`_ `` to see the available methods and how to implement them.

CHAPTER

FOUR

GUI TOOLKIT

4.1 Overview

4.1.1 Widgets

Widgets are the basic building blocks of graphical user interfaces (GUIs) in qooxdoo. Each GUI component, such as a button, label or window, is a widget and can be placed within an existing user interface. Each particular type of widget is provided by a corresponding subclass of Widget, which is itself a subclass of LayoutItem.

`Widget` can be subclassed with minimal effort to create custom widgets. The entire layout handling and children handling in this class is only available as "protected". It is possible to add some public API as needed.

Another framework class which extends `LayoutItem` is Spacer. A spacer is an empty area, which may be used as a temporary placeholder that is to be replaced later, or explicitly as a flexible part in certain dynamic UI designs.

To structure an interface it is common to insert widgets into each other. Each child is displayed within the screen area occupied by its parent. The hierarchical structure is also used to hide or show specific areas. This means for instance, that hiding a parent hides its children as well. Another example would be when a widget is being disposed, all the child widgets it contains are automatically being disposed as well.

4.1.2 Composites

As mentioned a few sentences above the normal `Widget` does not have public methods to manage the children. This is to allow the normal Widget to be used for inheritance. To allow the creation of structures in applications, the `Composite` was created.

Composite extends `Widget` and publishes the whole children and layout management of the `Widget` to the public. Typically it is used as a container for other widgets. Children can be managed through the methods `add()`, `remove()`, etc. In application code Composites are used to structure the interface.

4.1.3 Roots

A special category of widgets are the root widgets. These basically do the connection between the classic DOM and the qooxdoo widget system. There are different types of roots, each individually tuned for the requirements in the covered use case.

First of all every application developer needs to decide if an application should be standalone e.g. working with a minimal set of classic HTML or will be integrated into an maybe full-blown web page. Developers of an standalone application normally have no problem to give the control to the toolkit (maybe even enjoy it to give away this responsibility), but this would not work for integrating qooxdoo into an existing web page layout.

A standalone application normally only uses a really slimmed down set of HTML (in fact the file only functions as a wrapper to load the application code). It normally does not include any CSS files and often comes with an empty body element. In fact even simpler elements like headers, footers etc. are created using widgets (so they may benefit from typical qooxdoo features like internationalisation, theming etc.).

- Application: Build full-blown application from scratch. Target audience are developers of a completely qooxdoo based application.
- Page: Build applications as isles into existing content. Ideal for the more classic web developer. Needs to bring in know how of HTML & CSS for non-qooxdoo content.

Both roots are attached directly to the document. The `Application` is automatically stretched to the full size of the window and this way allows to position elements in relation to the right or bottom edge etc. This is not possible using the `Page` root.

The instantiation of the required root widget is normally nothing the developer has to do. It is done by the application class the developer chooses to extend. The next chapter will explain the concept behind applications in detail.

As even the `Page` root is attached to the document it would be still not possible to place children into a specific existing column or box into the existing layout. However the developer of the web page may use any number of optional isles to insert content into an existing layout (built with classic HTML markup). The isles are named Inline. They need an existing DOM element to do their work (maybe using some type of `getElementById`). The reason for the overall need, even when working with these isles, for the `Page` root is that all dynamically floating elements like tooltips, menus, windows etc. are automatically placed into this root. This makes positioning of such elements a lot easier.

4.1.4 Applications

The application is the starting point of every qooxdoo application. Every qooxdoo application should also come with a custom application class. The application is automatically initialized at the boot phase of qooxdoo (to be exact: when all required JavaScript packages are loaded).

The first method each developer needs to get used to is the main method. It is automatically executed after the initialization of the class. Normally the method is used to initialize the GUI and to load the data the application needs.

There are different applications which could be used as a starting point for a custom application:

- Standalone: Uses the `Application` root to build full blown standalone qooxdoo applications
- Inline: Uses the `Page` root to build traditional web page based application which are embedded into isles in the classic HTML page.
- Native: This class is for applications that do not involve qooxdoo's GUI toolkit. Typically they make only use of the IO ("Ajax") and BOM functionality (e.g. to manipulate the existing DOM).

4.2 Widgets Introduction

4.2.1 Widget

This is the base class for all widgets.

Features

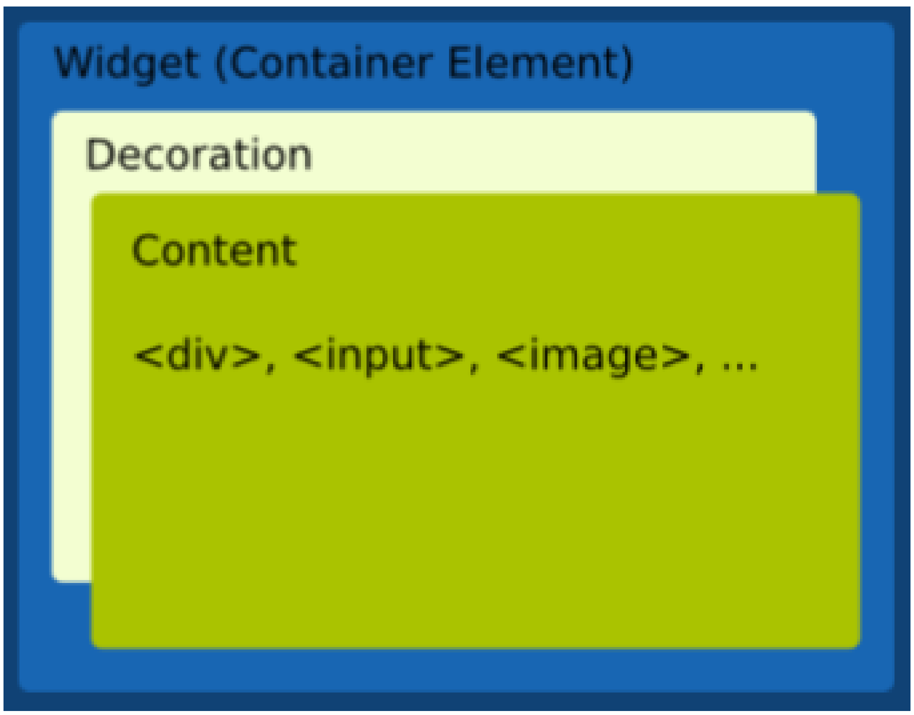
- Integration with event system
- Focus handling

- Drag and drop
- Auto sizing
- Theming
- Tool tips
- Context menus
- Visibility handling
- Sub widget management

Description

The widget is the base class for all qooxdoo widgets. It contains the widget system's core functionality.

Diagram



A widget consists of at least three HTML elements. The container element, which is added to the parent widget, has two child Elements: The "decoration" element and the "content" element. The decoration element has a lower z-Index and contains markup to render the widget's background and border using an implementation of `qx.ui.decoration.IDecorator`. The content element is positioned inside the "container" element to respect paddings and contains the "real" widget element.

Demos

There are no explicit widget demos since the widget is typically sub classed.

API

Here is a link to the API of the Widget:
qx.ui.core.Widget

4.2.2 Basic Widgets

Note: This chapter introduces some of the widgets found in qooxdoo. For a full list of widgets, please refer to the *Widget Reference*.

Labels

Labels are one of the basic building blocks in applications. The qooxdoo Label supports two modes: One which combines simple single line text content with the possibility to automatically render an ellipsis in cases where not enough room is available. This is often the best choice for all types of simple labels and is the default mode in qooxdoo. Through technical restrictions it is not possible to insert HTML in a so-configured instance. The other mode allows rich content (HTML) and adds the option for multi-line content together with an advanced mechanism called *Height4Width* which automatically re-wraps content based on the available width. This mode however cannot handle automatic ellipsis (which makes less sense in multiline labels, but is also not technologically possible).

More details: *Label*

Images

The second building block of applications. The image class in qooxdoo is quite sophisticated. PNG transparency is available in all browsers. Image data (e.g. format and dimension) is automatically pre-cached by the build system and distributed to the application for optional performance (avoiding page reflow during application startup for example).

This data also makes it possible to allow semi-automatic image sprites, a feature which becomes more important in larger applications. Image sprites combine multiple images (perhaps even with multiple states) in a single image instance. Only the relevant part is shown, all other states or images are cropped. This has positive effects on the latency (e.g. number of HTTP requests needed) and also improves the runtime performance (switching a state in an image sprite is much faster than replacing the source of an image instance). Image sprites can be introduced in any application at any time without changing the application code. The original image path is automatically interpreted as a clipped image source with the needed offsets. Please note that this feature largely depends on qooxdoo's tool chain which is required to generate the image data for the client.

A major restriction of this technology is that the options to resize images on the client side are crippled (the normal image is rendered through a background-image definition and allows no stretching at all). The alternate mode renders the image using a normal image element. This is a good alternative whenever a part of the application depends on this scaling feature but should not be used unless necessary.

More details: *Image*

Atoms

Atoms have been in qooxdoo for quite some time now. Basically, this widget combines an Image with a Label and allows some alignment options for them. Both content types are optional and toggle-able. The Atom supports shrinking like the Label while keeping the image intact. Atoms are used by many higher level widgets like Buttons (in Tab Views, Toolbars, ...) or List Items etc.

More details: *Atom*

Buttons

The Button is basically an Atom with some additional events. All relevant rendering features are already provided by the Atom. Several variants of the Button are available: Repeat, Radio or Toggle Button.

The Button can be connected to a Command (a class to work with key bindings etc.) and fires an `execute` event when clicked (or activated via the keyboard). The Repeat Button fires the `execute` event in an interval while being pressed. The Toggle Button (which toggles between checked and unchecked) is an exception to this and fires a `change` event on each transition of the `checked` property.

More details: *Button*

Text Fields

The Text Field is one of the most commonly used form elements. It fires two events: The `input` event is fired on every keystroke or other type of text modification. This event fires "live", i.e. whenever a modification is made. If the application does not need this level of detailed information, it should use the `change` event which fires after the modification is done, typically after the field has lost focus.

The Text Field supports basic label alignment to `left`, `center` or `right`. Preventing user inputs is possible through the property `enabled` or `readOnly`. Disabling a widget greys it out and makes it unresponsive for all types of interaction while `readOnly` only prevents the modification of the value and normally has no special visual indication when enabled.

More details: *TextField*

Popups

Popups and Tooltips are comparable in some way. Both are rendered above other content (while tooltips are even above Popups). Both widgets are automatically inserted into the application root widget (can be overridden when needed).

Popups may be used for notification panels or a type of modal sub dialog. Basically they are just a container (with a configurable layout) which lays above normal content.

By default, popups are automatically hidden if the user interacts with some other part of the application. This behavior is controllable through the `autoHide` property. Popups are automatically moved back inside the viewport. In fact, it is not possible to place Popups outside the viewport (not even partly). This behavior makes sense in almost every case and improves the usability of popups in general.

With `bringToFront` and `sendToBack` the popups' zIndex can be controlled in relation to other visible popups.

More details: *PopUp*

Tooltips

Tooltips are basically Popups with an Atom in them. But Tooltips improve on many of the features of the normal Popup. The automatic positioning support as mentioned for the Popups supports offsets as well and automatically moves the Tooltip to the best possible side in relation to the mouse cursor's position.

Although it's generally not necessary, every popup can be configured with an individual timeout. This is useful when building different type of tooltips e.g. to display system notifications etc.

More details: *ToolTip*

4.2.3 Interaction

Register listeners

To register listeners to a widget or other qooxdoo object just call `addListener()` with the given event type and callback method on them. The method will be executed every time the event occurs. Some types of events will bubble up the parent widget chain (such as mouse events, ...) while others are only fired on the original object (e.g. property changes, ...). A typical registration might look like this:

```
obj.addListener("changeColor", this._onChangeColor, this);
```

The first parameter is the name of the event. The events supported by an object are listed in the API documentation of each class in the "Events" section. The second argument is a pointer to a function to call. The function can also be defined inline (in a closure). The third argument defines the context in which the function is executed. This argument is optional and defaults to the object which is listened to, e.g. a listener on a button will call a function on the button.

The method is called with the event object as the first and only argument. The event object contains all information about the target and state of the event and also contains some other useful data: Mouse events may contain mouse coordinates while focus events may contain the focused element. Data events typically contain the current value of the data field listened to.

Please note that event objects are automatically pooled after their dispatch. This is mainly for performance reasons; event objects are reused during the application runtime. That's why keeping references to event instances is not a good idea! If some of the data is needed later during the application runtime it is best to store the actual data and not the event object, e.g. store the coordinates instead of the mouse event object.

Event Phases

In the browser most user input events like mouse or keyboard events are propagated from the target element up to the document root. In qooxdoo these events bubble up the widget hierarchy. This event propagation happens in two phases, the capturing and the bubbling event phase. The last parameter of the `addListener(type, listener, context, capture)` method defines whether the listener should be attached to the capturing (`true`) or bubbling (`false`) phase.

In the capturing phase, the event is dispatched on the root widget first. Then it is dispatched on all widgets down the widget tree until the event target is reached. Now the event enters the bubbling phase. In this phase the event is dispatched in the opposite direction starting from the event target up to the root widget.

Most of the time only the bubbling phase is used but sometimes the capturing phase can be very useful. For example a capturing listener for "mousedown" events on the root widget is guaranteed to receive every "mousedown" event even if the target widget calls `stopPropagation()` on the event. Further it can be used to block events from sub widgets.

Mouse Events

qooxdoo supports all the typical mouse events: `mousedown`, `mouseup`, `click` and `dblclick` as well as `mouseover` and `mouseout`. For most action-related widgets `execute` is the better choice than `click` (see the *section about basic widgets*). All these events behave identically in all supported browsers, even the sequence in which they are fired is identical. All of them come with a usable `target` and sometimes even with a `relatedTarget` for `mouseover` and `mouseout` events.

Every mouse event propagates the screen (e.g. `getScreenLeft()`), document (e.g. `getDocumentLeft()`) or viewport (e.g. `getViewportLeft()`) coordinates through the available getters. The `getWheelDelta()` delta method provides information about the scroll amount of a `mousewheel` event. Some widgets like Spinners or SelectBoxes make use of this event already.

During every mouse event it is possible to check the status of modifier keys through the methods `isCtrlPressed()`, `isAltPressed()` or `isShiftPressed()`. The pressed button can be detected by calling one of the methods `isLeftPressed()`, `isMiddlePressed()` or `isRightPressed()` on the mouse event.

See the API documentation of the MouseEvent for a full list of all available methods.

Event Capturing

Usually only the widget underneath the mouse cursor will receive mouse events. This can be a problem in drag operations where the mouse cursor can easily leave the dragged widget. This issue can be resolved in qooxdoo by declaring this widget a capturing widget using the widget's `capture()` method.

If a widget is a capturing widget, all mouse events will be dispatched on this widget, regardless of the mouse cursor's position. Mouse capturing is active until either a different widget is set to capture mouse events, the browser loses focus or the user clicks the left mouse button. If a widget loses its capture state a `losecapture` event is dispatched on the widget.

Internally, qooxdoo uses mouse capturing in menus, split panes or sliders for example.

Keyboard Support

DOM3-like event handling was the prototype for qooxdoo's key event support. This means that key identifiers can be used (instead of un-unified key codes) which is much more comfortable than what is known from most web application frameworks. Basically each key on the keyboard has a name like `Ctrl`, `Shift`, `F3` or `Enter`. A complete list of all supported keys is available in the API documentation.

All the typical key sequence events `keyup`, `keydown` and `keypress` support the key identifier. The `keypress` event is repeated during the time the key is pressed. That's why `keypress` is the best candidate for most action related keyboard events. Only use `keyup` and `keydown` when you *really* depend on the status of the key.

To handle character inputs e.g. on text boxes, there is a special `keyinput` event which has nice unified accessors, `getChar()` and `getCharCode()`, to detect the pressed character. This even automatically respects the effects modifier keys have, supporting e.g. German umlauts. The API lists all available methods of the KeyInput event.

Working with Commands

Commands (API) are used to bundle a command to be used by multiple buttons. They can also be used to define a global shortcut to be used for this action.

Creating new commands is as easy as it can be. A shortcut can simply be defined through the constructor, e.g.:

```
var find = new qx.event.Command("Ctrl+F");
find.addListener("execute", this._onFind, this);
```

The command can easily be attached to many types of Buttons etc. Some of them, like the `MenuButtons`, automatically display the configured shortcut as well. As seen above, the Commands also make use of the key identifiers.

Focus Handling

Good keyboard support also means good focus support. One major feature is the seamless integration between DOM focus handling and qooxdoo's focus handling. Both system communicate with each other. This makes it possible to integrate qooxdoo into normal web pages while still supporting the advanced focus features qooxdoo has to offer in qooxdoo-powered isles.

Focus handling in qooxdoo also means sophisticated support for the `Tab` key. While qooxdoo can also use the functionality provided by the browser, it adds its own layer for tab focus handling by default. This layer supports focus roots: A focus root is basically a widget which manages its own tab sequence. This is frequently used for many types of windows inside complex applications: Instead of leaving the window when reaching the last of its child widgets, the focus is moved back to the first child widget. The tab handling in qooxdoo is based on coordinates of each widget on the screen. It follows the visible structure and not the internal application (or even markup) structure. This is often seen as a huge benefit as it improves the usability of such applications out-of-the-box. It is also possible to define a `tabIndex` on widgets which should be reachable in a static hard-coded way. It is not advisable to use this feature too much. The automatic handling works quite well out of the box without hard-wiring every widget to a specific tab position.

To make a widget focusable just enable the property `focusable` (API) on it. For most widgets, this will also means that the widget is reachable using the `Tab` key, but this depends on the widget's implementation of the method `isTabable()`.

Every widget can function as a focus root. To register a widget as a focus root just call the method `addRoot()` of the `FocusHandler` like this:

```
qx.ui.core.FocusHandler.getInstance().addRoot(myWidget);
```

Activation is related to focus. While focus is limited to widgets which are marked as `focusable`, any widget can be activated. Usually, the activation moves around while clicking on widgets (during the `mouseup` event). The focus is applied to the next focusable parent while the activation directly happens on the widget that was clicked on. Activation is mainly used for keyboard support (key events start bubbling from the active widget). Compared to the focus, there is no visual highlighting for this state. To change the currently focused or active widget just call `focus()` or `activate()`:

```
myInputField.focus();
```

The properties `keepFocus` and `keepActive` are targeted more towards advanced users and developers of custom widgets. Both prevent the focus or active state from moving away (from the widget that currently has it) to the widget which has the specified property disabled. This is appropriate for complex widgets like a ComboBox where the activation should be kept on the ComboBox itself when selecting items from the dropdown list.

4.2.4 Resources

Resources comprise images, icons, style sheets, Flash files, helper HTML files, and so forth. The framework itself provides many icons and some other useful resources you can use right away in your application without any customization. This article however explains how to specify and use custom resources for your application.

Technical overview

Resources live in the `source/resource/<namespace>` subtree of each library. You explicitly reference a resource in your application code by just naming the path of the corresponding file **under** this root (This is also referred to as the **resource id**).

So if there is a resource in your "myapp" application under the path `myapp/source/resource/myapp/icons/tray.png` you would refer to it in your application code with `myapp/icons/tray.png`.

To find the corresponding file during a build, qooxdoo searches all those paths of all the libraries your application is using. The first hit will be regarded as the resource you want to use. (During the generation of a `build` version of your app, these resource files will be copied to the `build` folder, so your build version will be self-contained).

The libraries are searched in the order they are declared in your config.json file. This usually means that your own resource folder comes first, then the framework's resource folder, and then the resource folders of all further libraries you have included. This way, you can *shadow* resources of like names, e.g. by adding a file `qx/static/blank.gif` under your source/resource folder you will shadow the file of the same resource id in the framework.

Declaring resources in the code

You have to declare the resources you wish to use in your application code in an `#asset` compiler hint near the top of your source file.

```
/* ***

#asset(myapp/icons/16/folder-open.png)

*/
```

This is essential, since these hints are evaluated during the compile step, which searches for the corresponding files, generates appropriate URIs to them and copies them to the `build` folder.

Instead of adding meta information for each individual resource, you may as well use simple (shell) wildcards to specify a whole set of resources:

```
/* ***

#asset(myapp/icons/16/*)

*/
```

This is all you need to configure if your application code uses any of the icons in the given folder.

Using resources with widgets

Once you've declared the resource in your code, you can equip any compatible widget with it.

Here's an example:

```
var button = new qx.ui.form.Button("Button B", "myapp/icons/16/folder-open.png");
```

Using qooxdoo icons with widgets

If you want to use some of the icons as resources that are part of the icon themes that come with qooxdoo, there are the following three ways to do so:

1. Copy the icons you are interested in from the original location in the qooxdoo framework to the local resource folder of your application. You are now independent of the qooxdoo icon theme folders and can manage these icons as you would any other custom images.
2. Use a fully-qualified path that points to the qooxdoo resource folder. This solution would contain the icon theme's name explicitly.
3. Use a macro to get the icons from the current theme. This would allow for a later change of icon themes at the config file level, without the need to adjust any resource URIs in your application code. The *Generator documentation* explains how to declare these macros.

```
/*
#asset(myapp/icons/16/utilities-dictionary.png)
#asset(qx/icon/Oxygen/16/apps/utilities-dictionary.png)
#asset(qx/icon/${qx.icontheme}/16/apps/utilities-dictionary.png)
*/
```

```
...

var button1 = new qx.ui.form.Button("First Button", "myapp/icons/16/utilities-dictionary.png");
var button2 = new qx.ui.form.Button("Second Button", "qx/icon/Oxygen/16/apps/utilities-dictionary.png
var button3 = new qx.ui.form.Button("Third Button", "icon/16/apps/utilities-dictionary.png");
```

When you use the third method above and you do not use the *Modern* theme, you must edit `config.json` in order to have the meta theme's icons and the explicitly given icon theme in sync:

```
{
  "name"    : "myapp",

  ...

  "let" :
  {
    "APPLICATION"  : "myapp",
    ...
    "QXTHEME"      : "qx.theme.Classic",
    "QXICONTHEME"  : ["Oxygen"],
    ...
    "ROOT"         : "."
  }
}
```

Obtaining the URL for a resource

To obtain a URL for a resource, use the ResourceManager:

```
var iframe = new
qx.ui.embed.Iframe(qx.util.ResourceManager.getInstance().toUri("myapp/html/FAQ.htm"));
```

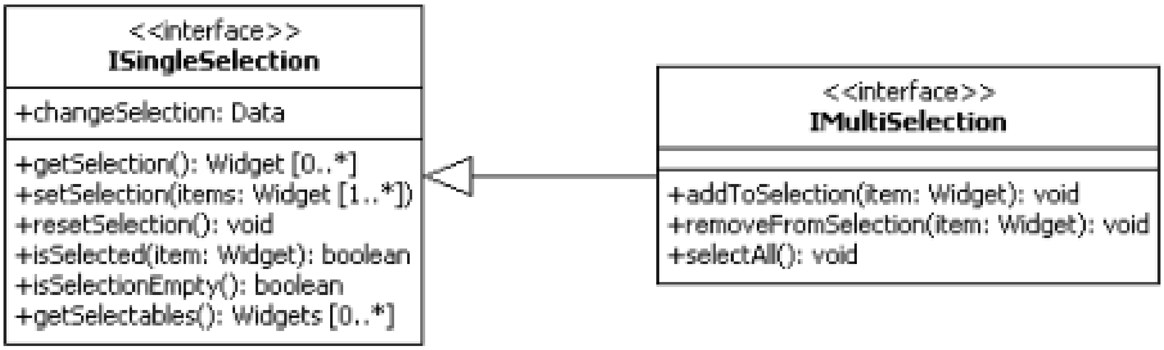
4.2.5 Selection Handling

The framework contains several widgets which support selection handling. These are divided into widgets that support `Single Selection` and others that support `Multi Selection`. A widget which supports multi selection also supports single selection.

Here is a list of widgets which support single and/or multi selection:

- Multi Selection:
  - Tree (API)
  - List (API)
- Single Selection:
  - SelectBox (API)
  - RadioGroup (API)
  - TabView (API)
  - Stack (API)

Selection Interfaces



Event

Both selections fire a `changeSelection` event if the selection has changed. Listeners can register with the event to be notified about the changes. The event contains an array with the newly selected widgets. If the array is empty, that means no widgets are selected.

```
list.addListener("changeSelection", function(e)
{
  var selection = e.getData();
  for (var i = 0; i < selection.lenght; i++) {
    this.debug("Selected item: " + selection[i]);
  }
}, this);
```

Selection Methods

The `ISingleSelection` interface specifies the methods for single selection handling. Since the methods of the single selection interface are re-used, the `IMultiSelection` only extends the interface with methods for multi selection handling.

Re-using the methods requires a uniform handling for setting and getting the current selection. This has been achieved by using an array for the selection handling, see `setSelection` and `getSelection`.

Single Selection

The listed single selection widgets above implement the `ISingleSelection`. To implement the behavior they use the MSingleSelectionHandling mixin. This mixin offers the methods for selection handling and also initializes the manager for selection management.

The widget itself configures the mixin to allowing an empty selection or not. Dependent on the configuration, `resetSelection` clears the current selection (empty array) or selects the first selectable element.

User interactions (mouse and keyboard) are managed from the widget, which only calls the selection methods if the user interaction has an effect on the selection. So the selection management and the user interaction handling are separated. This is one thing that has changed with the new selection API.

Multi Selection

The multi selection implementation has hardly changed at all. The widgets supporting multi selection, also listed above, have already used a mixin called MSelectionHandling for selection handling. Like the mixin for the single selection, it offers the selection methods and initializes the selection manager. The mixin has only been changed to conform to the new `IMultiSelection` interface.

Selection Modes

Due to the small changes the configuration for the selection mode hasn't changed. The widgets also support the property `selectionMode` with these different modes:

- **single:** Only one element or none at all can be selected.
- **one:** Exactly one item is selected if possible. The first selectable item is selected per default.
- **multi:** Multiple items can be selected by using the modifier keys together with mouse or keyboard actions. This type also allows empty selections.
- **adaptive:** Easy Web-2.0 selection mode: multiple items can be selected without modifier keys. Empty selections are possible.

Note: *Multi* and *Adaptive* selections dealing with **selection ranges**, *Single* and *One* dealing with one **selected item**.

```
list.setSelectionMode("multi");
```

Selection Options

These options change the way a selection is created or modified. By default, items can be selected by holding down the mouse button and hovering them or by holding down the modifier key and pressing the arrow keys to traverse them.

- **Quick:** One item can be selected by hovering it (no need to click on it or hit keys) Only possible for the modes *single* and *one*.
- **Drag:** Multiselection of items through dragging the mouse in pressed states. Only possible for the modes *multi* and *additive*.

```
list.setDragSelection(true);
```

How to use the selection API

Single Selection

The example below shows how to use the single selection API. This example uses the SelectBox widget:

```
// creates the SelectBox
var selectBox = new qx.ui.form.SelectBox();
this.getRoot().add(selectBox, {top: 20, left: 20});

// registers the listener
selectBox.addListener("changeSelection", function(event) {
  this.debug("Selected (event): " + event.getData()[0].getLabel());
}, this);

// creates the items and select one of them
```

```
for (var i = 0; i < 10; i++)
{
  var item = new qx.ui.form.ListItem("ListItem" + i);
  selectBox.add(item);

  if (i == 5) {
    selectBox.setSelection([item]);
  }
}

this.debug("Selected (selectBox): " + selectBox.getSelection()[0].getLabel());
```

The output should be:

```
(1) Selected (event): ListItem0
(2) Selected (event): ListItem5
(3) Selected (selectBox): ListItem5
```

The SelectBox's implementation doesn't allow empty selections, so if the first item is added to the SelectBox it will be selected (1). (2) occurs due to the selection and (3) from `getSelection`.

Multi Selection

The next example uses the List widget:

```
// creates the List and sets the selection mode
var list = new qx.ui.form.List();
list.setSelectionMode("multi");
this.getRoot().add(list, {top: 20, left: 20});

// registers the listener
list.addListener("changeSelection", function(event) {
  this.debug("Selection (event): " + event.getData());
}, this);

// creates the items
for (var i = 0; i < 10; i++)
{
  var item = new qx.ui.form.ListItem("ListItem" + i);
  list.add(item);
}

// sets selection
list.setSelection([list.getChildren()[1], list.getChildren()[4]]);

this.debug("Selection (list): " + list.getSelection());
```

The output should look like this:

```
(1) Selection (event): qx.ui.form.ListItem[1p],qx.ui.form.ListItem[2a]
(2) Selection (list): qx.ui.form.ListItem[1p],qx.ui.form.ListItem[2a]
```

4.2.6 Drag & Drop

Drag & Drop is one of the essential technologies in today's applications. An operation must have a starting point (e.g. where the mouse was clicked), may have any number of intermediate steps (widgets that the mouse moves over during a drag), and must either have an end point (the widget above which the mouse button was released), or be canceled.

qooxdoo comes with a powerful event-based layer which supports drag&drop with full data exchange capabilities. Every widget can be configured to cooperate with drag&drop be it as sender (draggable), receiver (droppable) or both. A sender (drag target) can send data to any receiver (drop target).

You may like to see an example first:

- Drag&Drop for Lists

Basics

To enable Drag & Drop the properties draggable and droppable must be enabled on the specific widgets. For list type sources or targets it's often enough to make the top-level widget drag- or droppable e.g. the list instead of the list items.

```
var dragTarget = new qx.ui.form.List;
dragTarget.setDraggable(true);

var dropTarget = new qx.ui.form.List;
dropTarget.setDroppable(true);
```

The basic drag&drop should start working with these properties enabled, but it will show the no-drop cursor over all potential targets. To fix this one needs to register actions (and optionally data types) supported by the drag target. This can be done during the `dragstart` event which is fired on the drag target:

```
dragTarget.addListener("dragstart", function(e) {
  e.addAction("move");
});
```

The drop target can then add a listener to react for the `drop` event.

```
dropTarget.addListener("drop", function(e) {
  alert(e.getRelatedTarget());
});
```

The listener now shows an alert box which should present the identification ID (classname + hash code) of the drag target. Theoretically this could already be used to transfer data from A to B.

Data Handling

qooxdoo also supports advanced data handling in drag&drop sessions. The basic idea is to register the supported drag data types and then let the drop target choose which one to handle (if any at all).

To register some types write a listener for `dragstart`:

```
source.addListener("dragstart", function(e)
{
  e.addAction("move");

  e.addType("qx/list-items");
  e.addType("html/list");
});
```

This is basically only the registration for the types which could theoretically be delivered to the target. The IDs used are just strings. They have no special meaning. They could be identical to typical mime-types like `text/plain` but there is no need for this.

The preparation of the data (if not directly available) is done lazily by the `droprequest` event which will explained later. The next step is to let the target work with the incoming data. The following code block appends all the dropped children to the end of the list.

```
target.addListener("drop", function(e)
{
  var items = e.getData("qx/list-items");
  for (var i=0, l=items.length; i<l; i++) {
    this.add(items[i]);
  }
});
```

The last step needed to get the thing to fly is to prepare the data for being dragged around. This might look like the following example:

```
source.addListener("droprequest", function(e)
{
  var type = e.getCurrentType();

  if (type == "qx/list-items")
  {
    var items = this.getSelection();

    // Add data to manager
    e.addData(type, items);
  }
  else if (type == "html/list")
  {
    // TODO: support for HTML markup
  }
});
```

Support Multiple Actions

One thing one might consider is to add support for multiple actions. In the above example it would be imaginable to copy or move the items around. To make this possible one could add all supported actions during the `drag` event. This might look like the following:

```
source.addListener("dragstart", function(e)
{
  // Register supported actions
  e.addAction("copy");
  e.addAction("move");

  // Register supported types
  e.addType("qx/list-items");
  e.addType("html/list");
});
```

The action to use is modifiable by the user through pressing of modifier keys during the drag&drop process. The preparation of the data is done through the `droprequest` as well. Here one can use the action (call `e.getCurrentAction()` to get the selected action) to apply different modifications on the original data. A modified version of the code listed above might look like the following:

```
source.addListener("droprequest", function(e)
{
  var action = e.getCurrentAction();
```

```
  var type = e.getCurrentType();
  var result;

  if (type === "qx/list-items")
  {
    result = this.getSelection();

    if (action == "copy")
    {
      var copy = [];
      for (var i=0, l=result.length; i<l; i++) {
        copy[i] = result[i].clone();
      }
      result = copy;
    }
  }
  else if (case == "html/list")
  {
    // TODO: support for HTML markup
  }

  // Remove selected items on move
  if (action == "move")
  {
    var selection = this.getSelection();
    for (var i=0, l=selection.length; i<l; i++) {
      this.remove(selection[i]);
    }
  }

  // Add data to manager
  e.addData(type, result);
});
```

As known from major operating systems, exactly three actions are supported:

- `move`
- `copy`
- `alias`

which could be combined in any way the developer likes. qooxdoo renders a matching cursor depending on the currently selected action during the drag&drop sequence. The event `dragchange` is fired on the source widget on every change of the currently selected action.

Runtime checks

There are a few other pleasantries. For example it is possible for `droppable` widgets to ignore a specific incoming data type. This can be done by preventing the default action on the incoming `dragover` event:

```
target.addListener("dragover", function(e)
{
  if (someRunTimeCheck()) {
    e.preventDefault();
  }
});
```

This could be used to dynamically accept or disallow specific types of drop events depending on the application status or any other given condition. The user then gets a `nodrop` cursor to signal that the hovered target does not accept the data. To query the source object for supported types or actions one would call the methods `supportsAction` or `supportsType` on the incoming event object.

Something comparable is possible during the `dragstart` event:

```
source.addListener("dragstart", function(e)
{
  if (someRunTimeCheck()) {
    e.preventDefault();
  }
});
```

This prevents the dragging of data from the source widget when some runtime condition is not solved. This is especially useful to call some external functionality to check whether a desired action is possible. In this case it might also depend on the other properties of the source widget e.g. in a mail program it is possible to drag the selection of the tree to another folder, with one exception: the inbox. This could easily be solved with such a feature.

Drag Session

During the drag session the `drag` event is fired for every move of the mouse. This event may be used to "attach" an image or widget to the mouse cursor to indicate the type of data or object dragged around. It may also be used to render a line during a reordering drag&drop session (see next paragraph). It supports the methods `getDocumentLeft` and `getDocumentTop` known from the `mousemove` event. This data may be used for the positioning of a cursor.

When hovering a widget the `dragover` event is fired on the "interim" target. When leaving the widget the `dragleave` event is fired. The `dragover` is cancelable and has information about the related target (the source widget) through `getRelatedTarget` on the incoming event object.

Another quite useful event is the `dragend` event which is fired at every end of the drag session. This event is fired in both cases, when the transaction has modified anything or not. It is fired when pressing Escape or stopping the session any other way as well.

A typical sequence of events could look like this:

- `dragstart` on source (once)
- `drag` on source (mouse move)
- `dragover` on target (mouse over)
- `dragchange` on source (action change)
- `dragleave` on target (mouse out)
- `drop` on target (once)
- `droprequest` on source (normally once)
- `dragend` on source (once)

Reordering items

Items may also be reordered inside one widget using the drag&drop API. This action is normally not directly data related and may be used without adding any types to the drag&drop session.

```
reorder.addListener("dragstart", function(e) {
  e.addAction("move");
});
```

```
reorder.addListener("drop", function(e)
{
  // Using the selection sorted by the original index in the list
  var sel = this.getSortedSelection();

  // This is the original target hovered
  var orig = e.getOriginalTarget();

  for (var i=0, l=sel.length; i<l; i++)
  {
    // Insert before the marker
    this.addBefore(sel[i], orig);

    // Recover selection as it gets lost during child move
    this.addToSelection(sel[i]);
  }
});
```

4.2.7 Inline Widgets

This page describes how you can use qooxdoo widgets inside HTML-dominated pages. This use case is different from creating a regular, “standalone” qooxdoo application.

Target Audience

Integrating qooxdoo widgets into existing HTML pages could be interesting to all users who already have (many) existing pages, often some kind of “portal”, and therefore don’t want to transform these into a standalone rich Internet application (RIA).

Online Demos

Take a look at the online demos to see the use of inline widgets in action.

- Absolute positioning demo
- Page flow using Inline
- Dynamic resize for Inline
- Inline window

Set Up An Inline Application

An inline application is set up by using the `create-application` script described in the *Hello World* section. You just have to add the additional option `-t` with the value `inline` and you’re done.

```
/opt/qooxdoo-sdk/tool/bin/create_application.py -n myapp -t inline
```

Once executed you get a skeleton application which is ready-to-use to develop an inline application. The skeleton also demostrates the different integration approaches which are described in the next section.

Ways of Integration

There are basically two ways to integrate a qooxdoo widget into an existing HTML-dominated page:

- positioning a widget with absolute coordinates (maybe overlaying existing content)
- adding the widget within the page flow by using an existing DOM node as an isle

Which way you should choose depends on what you wish to achieve. Technically both share the same foundation.

Instead of using `qx.application.Standalone` as a base application class you need to extend from `qx.application.Inline` as a starting point. So basically your (empty) application looks like this:

```
qx.Class.define("myPortal.Application",
{
  extend : qx.application.Inline,

  members :
  {
    main: function()
    {
      this.base(arguments);

      // your code follows here
    }
  }
});
```

Absolute Positioning

Adding a widget to the page without regarding the page flow is a no-brainer. Just create the desired widget and add it to the application root. As the application root is an instance of qx.ui.layout.Basic you can only use `left` and `top` coordinates to position your widgets.

Note: Absolute positioning requires no existing DOM node in the target document.

```
qx.Class.define("myPortal.Application",
{
  extend : qx.application.Inline,

  members :
  {
    main: function()
    {
      this.base(arguments);

      // add a date chooser widget
      var dateChooser = new qx.ui.control.DateChooser();

      // add the date chooser widget to the page
      this.getRoot().add(dateChooser, { left : 100, top : 100 });
    }
  }
});
```

Page Flow

However, the former solution won't fit for e.g. a portal where the page is divided into several parts. In this case you won't have any absolute coordinates you could work with reliably.

To add widgets at certain locations inside the page you can create or reuse DOM nodes which act as islands where the qooxdoo widgets live in regard to the page flow.

Note: You need to define specific DOM nodes in your document which act as islands for the qooxdoo widgets.

```
qx.Class.define("myPortal.Application",
{
  extend : qx.application.Inline,

  members :
  {
    main: function()
    {
      this.base(arguments);

      // create the island by connecting it to the existing
      // "dateChooser" DOM element of your HTML page.
      // Typically this is a DIV as in <div id="dateChooser"></div>
      var dateChooserIsle = new qx.ui.root.Inline(document.getElementById("dateChooser"));

      // create the date chooser widget and add it to the inline widget (=island)
      var dateChooser = new qx.ui.control.DateChooser();
      dateChooserIsle.add(dateChooser);
    }
  }
});
```

4.2.8 Custom Widgets

Most widgets are built using a combination of pre-existing, more basic widgets. This is also true for custom widgets made for a specific application or as an extension to the existing feature set of qooxdoo.

Inheritance Structure

A more complex widget usually extends the base class `qx.ui.core.Widget`. A widget can manage children using a set of protected methods. Extending from a richer widget often has the side effect that the final class contains APIs which do not make sense in the derived class anymore. Also be sure not to extend from `Composite` or a widget based on this class. This is mainly because it has public methods for the normally internal layout and children handling and would propagate all the internal information to the outside when children are added or the layout is modified by the derived class.

A good example: Most rich text editors implemented in JavaScript make use of an iframe. One could imagine using the `Iframe` class as a base to build such a component. The problem is that most of the methods and properties like `setSource` or `reload` do not make a lot of sense on an editor component. It's better to embed the needed widgets into the outer widget to hide their functionality in the custom class.

The qooxdoo `Spinner` for example extends the `Widget` as well and adds a `TextField` and two `RepeatButton` instances. The layout is done by a Grid layout. All the children and the chosen layout are hidden from the outside. There are no public accessors for the layout or the children. This makes sense as no one is interested in the children of a `Spinner` widget. These methods would also mean a lot of bloat added to the API of such an widget.

Setup Content

The following methods may be used to manage children:

- _getChildren
- _add, _addAt, _addBefore, _addAfter
- _remove, _removeAt, _removeAll

It is possible to use any layout available. To set up the layout just use _setLayout. To access it afterwards use _getLayout.

For details refer to the API documentation of qx.ui.core.Widget.

Child Controls

qooxdoo supports a mechanism called child controls. A child control is a widget as part of another widget. Child controls were introduced to have a common way of accessing these controls and to make it easy to refine them when a class should be extended. Each child control is accessible using an identifier which is basically a string. By convention these strings are all lower-case und use dashes to structure complex identifiers. Typical identifiers are button, icon or arrow-up. Never slashes / as this might conflict with the appearance system.

Instances for the supported child controls are created dynamically as needed. A widget developer just needs to override the method _createChildControlImpl, let the method work on the customized controls, and just call the super class method when the incoming ID is not supported. For example, such a method might look like:

```
_createChildControlImpl : function(id)
{
  var control;

  switch(id)
  {
    case "icon":
      control = new qx.ui.basic.Image;
      this._add(control);
      break;
  }

  return control || this.base(arguments, id);
}
```

Each child control should directly add itself to the parent. As mentioned before child controls are automatically created as needed. This basically means that if nobody asks for a specific child control it is never created or added. This is an important feature for dynamic widgets as it reduces the initial memory and CPU usage. A child control is always created when some code asks for it. This can happen through different methods:

- getChildControl(id, notcreate): Returns the child control with the given ID. May return null if the second argument is true. This is basically used to check if the child control has already been created and then apply something to it. In some more complex scenarios this makes sense, but it can be ignored for the moment.
- _showChildControl(id): Executes show() on the child control. This method also creates the control if that hasn't happened yet. It also returns the control so other properties can be applied to it.
- _excludeChildControl(id): Excludes the widget using exclude(). When the control is not yet created the function does nothing. The method has no return value.
- _isChildControlVisible(id): Returns true if the child control with the given ID is created and visible.

- `hasChildControl(id)`: Returns `true` if the child control with the given ID has been created.

Styling

Child controls are automatically supported by the appearance system. For every child control a selector is generated which starts with the first widget which is not a child control itself. Typical selectors look like:

- `spinner/up-button`
- `groupbox/legend`
- `tree-item/icon`

As a container for child controls may be a child control for another container as well, even more complex selectors are possible:

- `list/scrollbar-x/slider`
- `splitbutton/button/icon`

This means that even the deepest child control can be easily accessed by theme authors. Widget authors should define the styling of a widget in the appearance theme and not in the widget itself. The widget and the `_createChildControlImpl` method should only apply functional properties like `zIndex` or `tabIndex`, but no decorations, colors or fonts for example.

As mentioned, a key always starts with the appearance of the first widget which is not itself a child control. Appearance values of the inner widgets are ignored as long as they are used as a child control. Instead, the ID of the child control is used. The `/` is used to separate the child controls. All widgets added through user code start with their own appearance. For example, the items of the `List` widget have the appearance `list-item`. Their appearance key is also `list-item` and not `list/item`.

For details about styling please refer to *the theming article*.

HTML Elements

A normal qooxdoo widget consists of at least two HTML Elements (API). The first one is the container element which is the outer frame of each widget. The inner one is the content element which is the target for children added to the widget. The content element is also used for the iframe element of the `Iframe` widget and the image element of the `Image` widget. This means it may contain children or may be used by a native DOM element which does not allow any children.

There might be some other elements depending on the configuration:

- shadow: Placed into the container with negative offsets to be visible behind the original widget.
- decorator: Placed into the container with the same size as the container. Used to render all kinds of decorators.
- protector: Helper to fix certain hover issues when changing decorators during event sequences, e.g. hover effects.

For widget authors, the content element is normally the most important, followed by the container element. The other elements are quite uninteresting. It is good to know that they are there, but one typically has little to do with them.

Both elements are instances of `qx.html.Element` so they come with a cross-browser fixed API to apply styles and attributes to the DOM nodes. All of these things can be done without the DOM element needing to be created or inserted. For details on `qx.html.Element` please have a look at *the technical documentation*.

The elements are accessible through the functions `getContentElement()` and `getContainerElement()`, respectively. The elements are stored privately in each widget instance and are only accessible through these methods in derived classes.

Custom Elements

qooxdoo normally generates a bunch of styled `div` elements. Some widgets like iframes or images need other elements, though. Normally the only element which is replaced is the content element. To achieve this, the method `_createContentElement` needs to be overwritten. The overwritten method should create an instance of `qx.html.Element` (or a derived class), configure it with some static attributes or styles, and finally return it. For most natively supported types there exists a class which can be used already. In special cases the widget author also needs to write a special low-level class which is derived from `qx.html.Element`.

Working with Events

Events can be added to the HTML elements as well as to the child controls. The names of the methods assigned should follow the following names for convention.

- For the HTML elements use: `_onContentXXX` or `_onContainerXXX`
- For the child controls use: `_onIconXXX` or `_onFieldXXX` etc.

Where `XXX` stands for the name of the event or of the change that happens. This will result in names like `_onIframeLoad` or `_onContentInput`.

Anonymous Widgets

Anonymous widgets are ignored in the event hierarchy. This is useful for combined widgets where the internal structure does not have a custom appearance with a different styling from the enclosing element. This is especially true for widgets like checkboxes or buttons where the text or icon are handled synchronously for state changes to the outer widget.

A good example is the `SelectBox` widget where the `mouseover` event should affect the entire widget at once and not the different child controls of which it consists. So setting the child controls (in this case an `atom` and an `image` widget) to `anonymous` keeps these child control widgets from receiving any events and the event handling is done completely by the parent widget (the `SelectBox` itself).

4.2.9 Form Handling

The `qx.ui.form` package contains several classes for the construction of forms. Some widgets – like `Button`, `List` or `TextField` – may look familiar if you have worked with HTML before, but this package also contains more complex widgets that you may know from your operating system and/or native desktop applications (e.g. `Spinner`, `Slider` or `DateField`).

Idea

The idea of the form API is to make handling of form widgets as simple as possible, but also as generic as possible within the entire framework. There has been a thorough discussion on what would be the best solution and how to design a solid API. This is what we ended up with.

Demos

If you like to see some of qooxdoo's form management in action, take a look at the following samples in the demo browser:

Widgets

- All form widgets
- All form widgets with invalid states

Validation and Resetting

- Synchronous and asynchronous form validation
- Validation on different pages

Rendering

- Single column form
- Double column form
- Single column form using placeholders
- Custom form layout

Data Binding

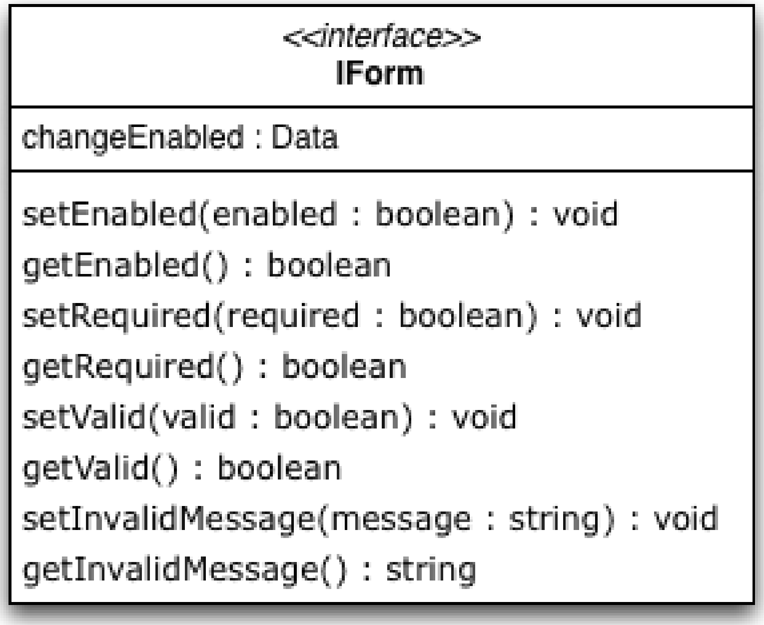
- Manual form binding
- Form Controller

Interfaces

The entire form API is defined by a couple of interfaces. These interfaces contain the most important methods and events for the form widgets. The following listing shows the interfaces, their purpose and how you can benefit from them.

Form

The interface `qx.ui.form.IForm` defines a set of methods and events for every visible form widget. It contains the listed events and methods.

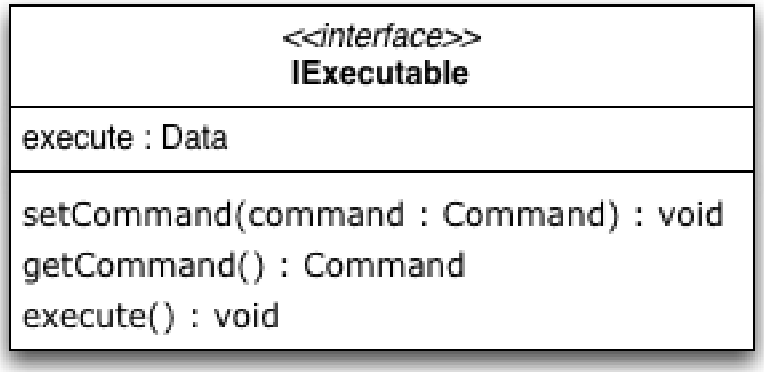


As you can see, the interface defines accessors for four different properties.

- The enabled property is usually inherited from the widget class and is used to deactivate a form element.
- The required property is just a boolean flag signaling that the form widget is required. This can be used by some kind of form manager or parent widget to display the status of the widget.
- The valid property is a boolean flag containing `true` if the content of the widget is valid, but the form widgets do not have any kind of code to set this property. It needs to be set from outside. If it is set to `false`, the appearance will change automatically to properly signal the invalid state.
- The invalidMessage property should contain a message which will be shown in a tooltip if the valid flag is set to `false`. If no message is given, no tooltip will appear.

Executable

The `qx.ui.form.IExecutable` interface defines the essential components for all executable widgets. The best example for an executable widget is a button. It defines the following events and methods.

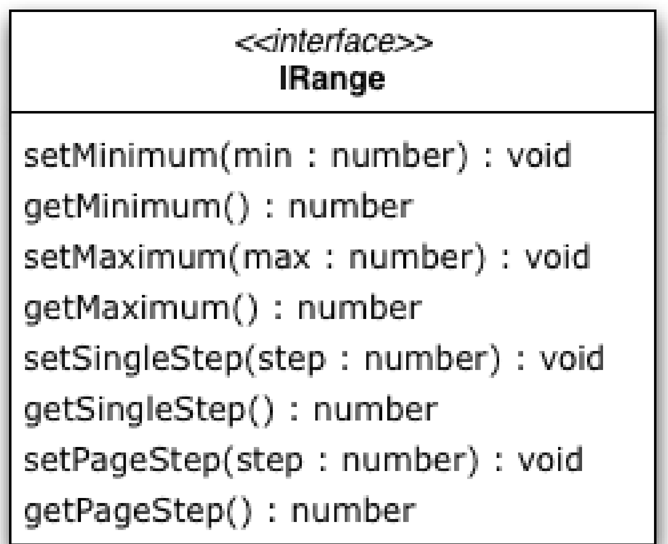


As you can see, the interface defines accessors for only one property.

- The command property can take a `qx.event.Command`. The execute method executes the given command.

Range

The `qx.ui.form.IRange` interface defines the essential components for all widgets dealing with ranges. It defines the following methods.



As you can see, the interface defines accessors for four properties.

- The minimum value of the range is defined by the Minimum property.
- The maximum value of the range is defined by the Maximum property.
- Each range has a single step value which is defined by the SingleStep property.
- Like the single step, there is a page step for every range which is defined by the PageStep property.

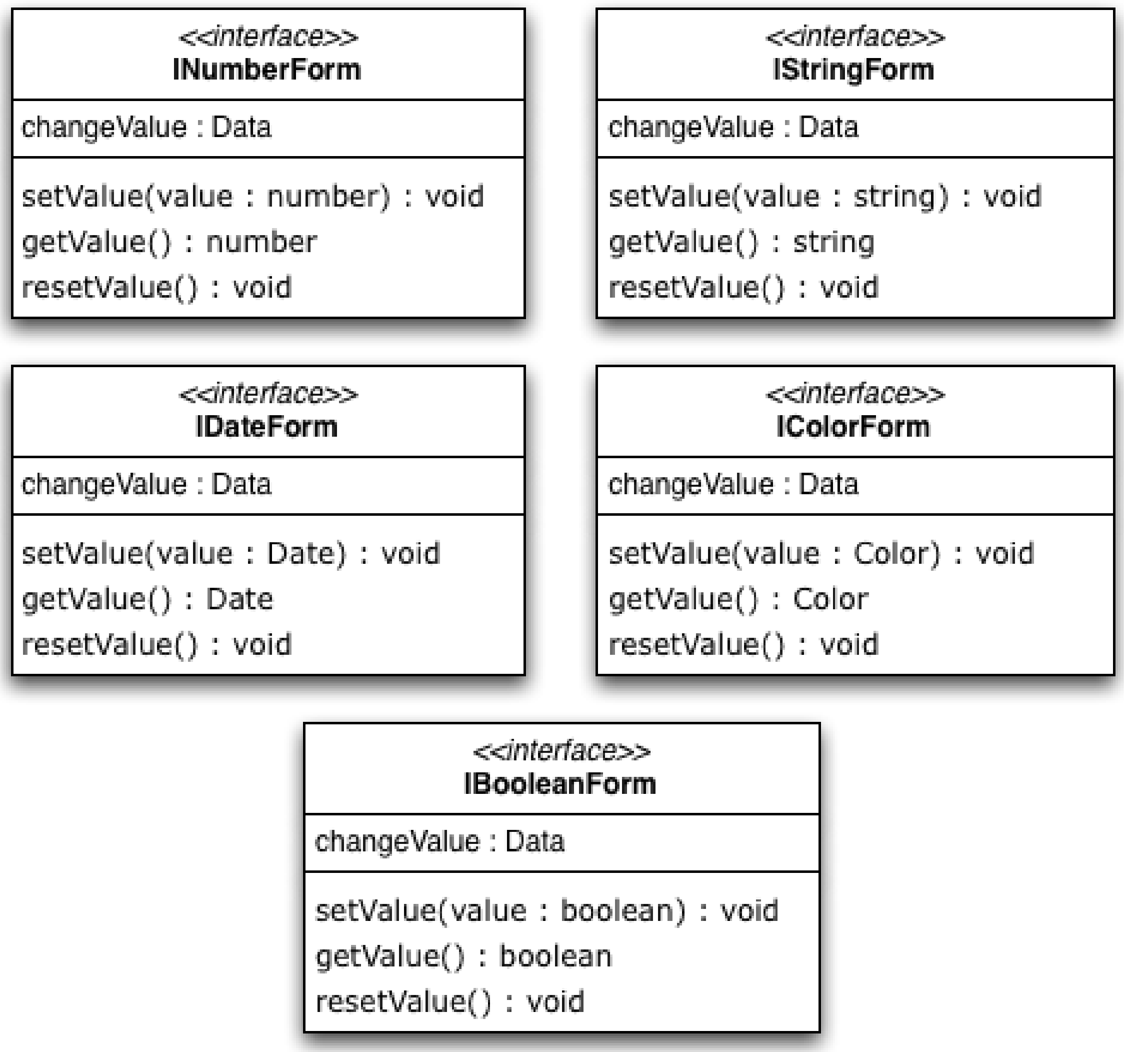
Number / String / Color / Date / Boolean

Each of the listed interfaces define the same methods and events. The only difference in the interfaces is - as the name says - the type of the data processed by the implementing widget. With that solution, we have the same API for every form widget but can still determinate which type of value the widget expects by checking for the different interfaces.

Interfaces

- Number : `qx.ui.form.INumberForm`
- String : `qx.ui.form.IStringForm`
- Color : `qx.ui.form.IColorForm`
- Date : `qx.ui.form.IDateForm`
- Boolean : `qx.ui.form.IBooleanForm`

The color interface takes a string which has to be formatted like the common colors in qooxdoo.

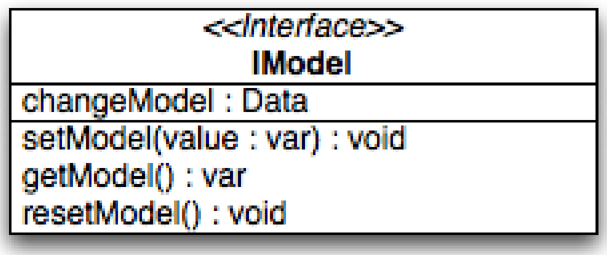


As you can see, the interface can be implemented with only one property.

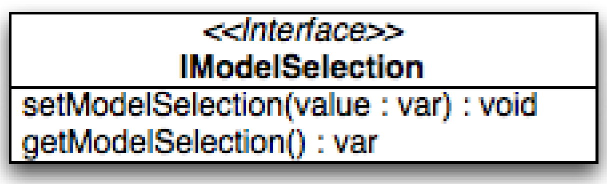
- The value property takes the value of the widget. This is for example a boolean in a checkbox widget or a string in a text field widget.

Model / ModelSelection

Most of the form items handling a selection had a value property in the old API. We replaced that with a model property since the the value property is used for user input values. The methods for accessing the model data are defined in an interface called `qx.ui.form.IModel`.



The model property can be used to store additional data which is represented by the widget. The data does not need to be a string like in the old value property. You can store references to objects, numbers, strings and so on. Accessing the model is very easy. Every widget containing a widget implementing the `qx.ui.form.IModel` interface has its own interface to access the current selected model.



As you can see in the diagram, you can get the currently selected model and also set the selection using the models.

Widgets

The following listing shows the form widgets and their corresponding interfaces. To see more details about a widget, take a look at the *widgets* documentation.

Sample Usage

The first example is a simple one, showing how to use two widgets implementing the `IStringForm` interface:

```
// create and add a textfield
var textfield = new qx.ui.form.TextField();
this.getRoot().add(textfield, {left: 10, top: 10});

// create and add a label
var label = new qx.ui.basic.Label();
this.getRoot().add(label, {left: 10, top: 40});
```

```
// set the text of both widgets
textfield.setValue("Text");
label.setValue("Text");
```

The second example shows how to react on a change in a widget implementing the `INumberForm` interface. The value of the slider will be shown as a label:

```
// create and add a slider
var slider = new qx.ui.form.Slider();
slider.setWidth(200);
this.getRoot().add(slider, {left: 10, top: 10});

// create and add a label
var label = new qx.ui.basic.Label();
this.getRoot().add(label, {left: 220, top: 10});

// add the listener
slider.addListener("changeValue", function(e) {
  // convert the number to a string
  label.setValue(e.getData() + "");
}, this);
```

The last example shows how to use the `IForm` interface and how to mark a widget as invalid:

```
// create and add a slider
var slider = new qx.ui.form.Slider();
slider.setWidth(200);
slider.setValue(100);
this.getRoot().add(slider, {left: 10, top: 10});
// set the invalid message
slider.setInvalidMessage("Please use a number above 50.");

// add the validation
slider.addListener("changeValue", function(e) {
  if (e.getData() > 50) {
    slider.setValid(true);
  } else {
    slider.setValid(false);
  }
}, this);
```

All examples work in the Playground application.

Migrating to the new API

There are some important topics you have to keep in mind if you want to migrate from the former Form API to the new one.

IFormElement

The previous form interface called `qx.ui.form.IFormElement` is deprecated now. Therefore, the name and value properties for storing string information for serialization are gone also. If you are using those constructs, you can instead use regular user data:

```
widget.setName("field1");  // old
widget.setUserData("name", "field1");  // new
```

This works identically to the old code. The HTML name property will not be set after the call in both cases.

CheckBox and RadioButton

Widgets like CheckBox or RadioButton had a `checked` property for their state. This property is deprecated and is now called `value`.

changeValue on List and SelectBox

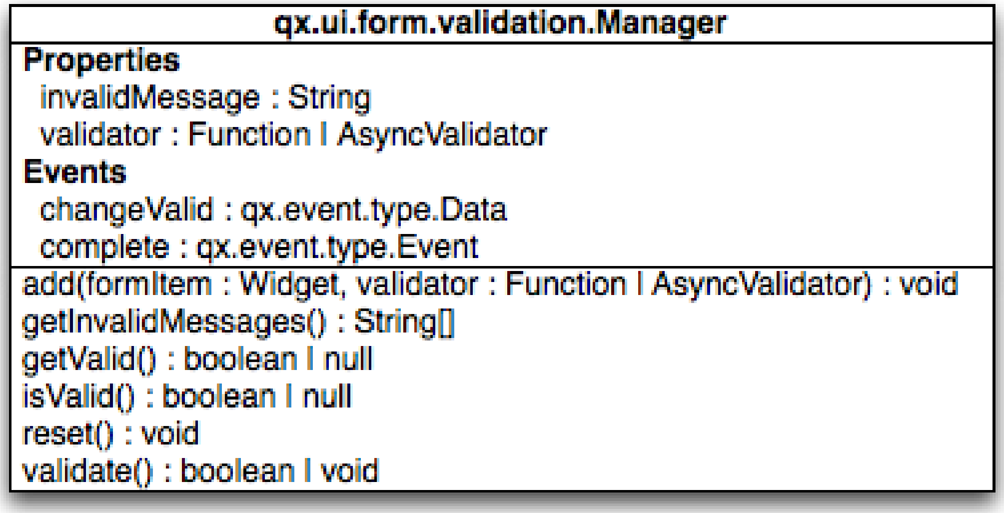
It was quite common to use the `changeValue` event of a `SelectBox` or `List` to handle a change of the selection. Due to the removal of `value`, the `changeValue` event has also been removed. Please use the `changeSelection` event instead.

Label

The former `content` property of the Label class has been renamed to make it consistent with the rest of the framework. So the new name is the same as in every other widget: `value`.

Validation

Form validation is essential in most of the common use cases of forms. Thats why qooxdoo supports the application developer with a validation component named `qx.ui.form.validation.Manager`. This manager is responsible for managing the form items which need to be validated. We tried to keep the API as minimal as possible but simultaneously as flexible as possible. The following class diagram shows the user API of the component.



The events, properties and methods can be divided into three groups:

- **Validation**
  - getValid()
  - isValid()
  - validate()
  - validator - property
  - complete - event
  - changeValid - event
- **Form Item Management**
  - add(formItem, validator)
  - reset()
- **Invalid Messages**
  - getInvalidMessages()
  - invalidMessage - property

The first part with which the application developer gets in contact is the add method. It takes form items and a validator. But what are form items?

Requirements

Form items need two things. First of all, a given form item must be able to handle an invalid state and must have an invalid message. This is guaranteed by the *IForm* interface already introduced. But that's not all: The manager needs to access the value of the form item. Therefore, the form item needs to specify a value property. This value property is defined in the *data specific form interfaces* also introduced above. So all widgets implementing the `IForm` interface and one of the value defining interfaces can be used by the validation. For a list of widgets and the interfaces they implement, take a look at the *widgets section* in this document.

Now that we know what the manager can validate, it's time to learn how to validate. In general, there are two different approaches in validation. The first approach is client side validation, which is commonly synchronous. On the other hand, server side validation is asynchronous in most cases. We will cover both possibilities in the following sections.

Synchronous

The following subsections cover some common scenarios of synchronous validation. See this code snippet as basis for all the examples shown in the subsections.

```
var manager = new qx.ui.form.validation.Manager();
var textField = new qx.ui.form.TextField();
var checkBox = new qx.ui.form.CheckBox();
```

Required Form Fields One of the most obvious validations is a check for a non-empty field. This can be seen in common forms as required fields, which are easy to define in qooxdoo. Just define the specific widget as required and add it to the validation manager without any validator.

```
textField.setRequired(true);
manager.add(textField);
```

The validation manager will take all the necessary steps to mark the field as invalid as soon as the validate method is invoked if the text field is empty.

Default Validator Another common use case of validation is to check for specific input types like email addresses, URLs or similar. For those common checks, qooxdoo offers a set of predefined validators in `qx.util.Validate`. The example here shows the usage of a predefined email validator.

```
manager.add(textField, qx.util.Validate.email());
```

Custom Validator Sometimes, the predefined validators are not enough and you need to create an application-specific validator. That's also no problem because the synchronous validator is just a JavaScript function. In this function, you can either return a boolean which signals the validation result or you can throw a `qx.core.ValidationError` containing the message to be displayed as an invalid message. The validation manager can handle both kinds of validators. The example here checks if the value of the text field has a length of at least 3.

```
manager.add(textField, function(value) {
  return value.length >= 3;
});
```

Validation in the context of the form All shown validation rules validate each form item in its own context. But it might be necessary to include more than one form item in the validation. For such scenarios, the manager itself can have a validator too. The example here demonstrates how to ensure that the text field is not empty if the checkbox is checked.

```
manager.setValidator(function(items) {
  if (checkBox.getValue()) {
    var value = textField.getValue();
    if (!value || value.length == 0) {
      textField.setValid(false);
      return false;
    }
  }
  textField.setValid(true);
  return true;
});
```

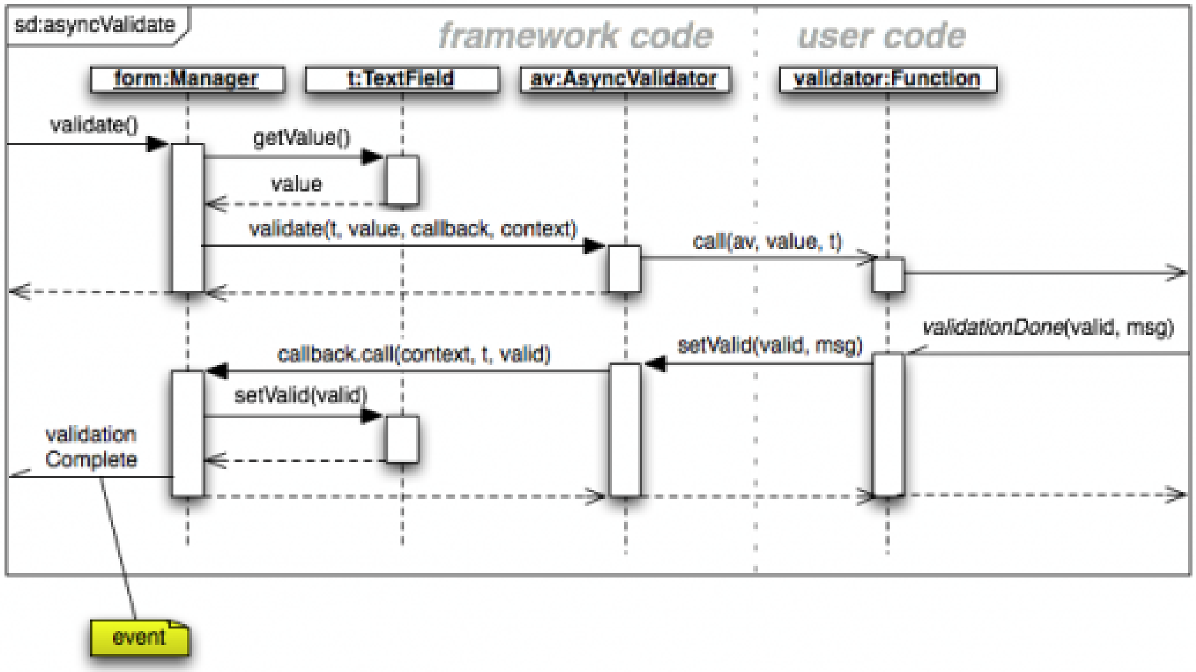
Asynchronous

Imagine a scenario where you want to check if a username is already taken during a registration process or you want to verify a credit card number. This type of validation can only be done by a server and not in the client. But you don't want the user to wait for the server to process your request and send the answer back. So you need some kind of asynchronous validation.

For all asynchronous validation cases, we need a wrapper for the validator, the `qx.ui.form.validation.AsyncValidator`. But that does not mean a lot work for the application developer. Just take a look at the following example to see the AsyncValidator in action.

```
manager.add(textField, new qx.ui.form.validation.AsyncValidator(
  function(validator, value) {
    // here comes the async call
    qx.event.Timer.once(function() {
      // callback for the async validation
      validator.setValid(false);
    }, this, 1000);
  }
));
```

The only difference to the synchronous case, at least from the application developer's point of view, is the wrapping of the validator function. Take a look at the following sequence diagram to get an insight on how the asynchronous validation is handled.



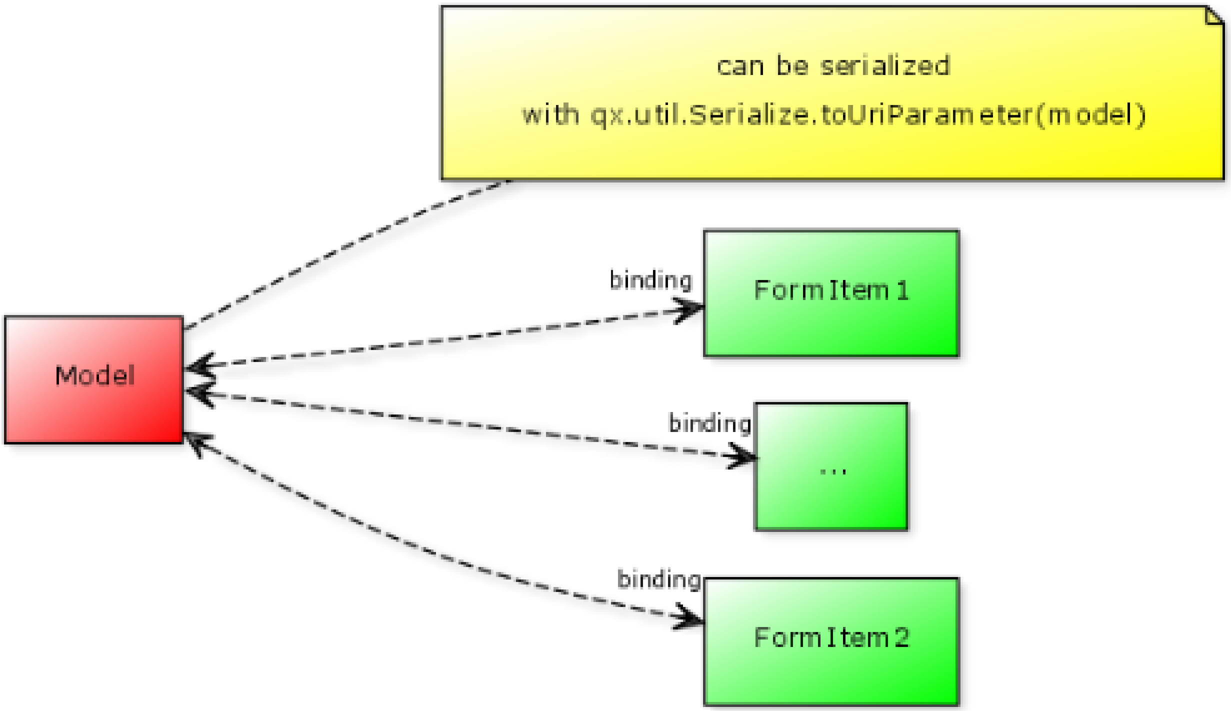
The asynchronous validation can not only be used for form items. Also, the manager itself can handle instances of the AsyncValidator as validator.

Serialization

Entering data into a form is one part of the process. But usually, that entered data needs to be sent to the server. So serialization is a major topic when it comes to forms. We decided not to integrate this in one form manager which would be responsible for both validation and serialization.

Idea

The main idea behind this was to ensure that it cooperates nicely with features like the form widgets and the corresponding data binding components. So we decided to split the problem into two different parts. The first part is storing the data held in the view components as a model. The second part takes that model and serializes its data. Sounds like *data binding*? It is data binding!



But you don't have to connect all these widgets yourself. qooxdoo offers an object controller which can take care of most of the work. But where do you get the model? Writing a specific qooxdoo class for every form sounds like a bit of overkill. But qooxdoo has a solution for that, too. The creation of classes and model instances is already a part of the data binding components and can also be used here. Sounds wierd? Take a look at the following common scenarios to see how it works.

Common Scenarios

The most common scenario is to serialize a number of form items without any special additions. Just get the values of the entire form and serialize them.

```
// create the ui
var name = new qx.ui.form.TextField();
var password = new qx.ui.form.PasswordField();

// create the model
var model = qx.data.marshal.Json.createModel({name: "a", password: "b"});

// create the controller and connect the form items
var controller = new qx.data.controller.Object(model);
controller.addTarget(name, "value", "name", true);
controller.addTarget(password, "value", "password", true);

// serialize
qx.util.Serializer.toUriParameter(model);
```

The result will be `name=a&password=b` because the initial values of the model are `a` and `b`.

This way, the serialization is separated from the form itself. So hidden form fields are as easy as it could be. Just add another property to the model.

```
var model = qx.data.marshal.Json.createModel(
  {name: "a", password: "b", c: "i am hidden"}
);
```

Keep in mind that you're creating a model with that and you can access every property you created using the default getters and setters.

You might be asking yourself "What if i want to convert the values for serialization? My server needs some different values...". That brings us to the topic of conversion. But as we have seen before, the mapping from the view to the model is handled by the data binding layer which already includes conversion. Take a look at the *data binding documentation* for more information on conversion.

Need something special? In some cases, you might want to have something really special like serializing one value only if another value has a special value or something similar. In that case, you can write your own serializer which handles serialization the way you need it.

Resetting

A third useful feature of a form besides validation and serialization is resetting the entire form with one call. Doesn't sound complicated enough that a separate class is needed. But we decided to do it anyway for good reasons:

- The validation manager is not the right place for resetting because it handles only the validation.
- The form widget, responsible for layouting forms, is a good place, but we don't want to force developers to use it if they just want the reset feature.

So we decided to create a standalone implementation for resetting called `qx.ui.form.Resetter`.

Like the task of resetting itself, the API is not too complicated. We have one method for adding items, and another one for resetting all added items.

How It Works

Technically, it's not really a challenge thanks to the new form API. You can add any items either having a value property defined by one of the *data specific form interfaces* or implementimg the *selection API* of qooxdoo. On every addition, the resetter grabs the current value and stores it. On a reset all stored values are set.

Sample Usage

The following sample shows how to use the resetter with three input fields: A textfield, a checkbox and a list.

```
// create a textfield
var textField = new qx.ui.form.TextField("acb");
this.getRoot().add(textField, {left: 10, top: 10});

// create a checkbox
```

```
var checkBox = new qx.ui.form.CheckBox("box");
this.getRoot().add(checkBox, {left: 10, top: 40});

// create a list
var list = new qx.ui.form.List();
list.add(new qx.ui.form.ListItem("a"));
list.add(new qx.ui.form.ListItem("b"));
list.setSelection([list.getSelectables()[0]]);
this.getRoot().add(list, {left: 10, top: 70});

// create the resetter
var resetter = new qx.ui.form.Resetter();
// add the form items
resetter.add(textField);
resetter.add(checkBox);
resetter.add(list);

// add a reset button
var resetButton = new qx.ui.form.Button("Reset");
resetButton.addListener("execute", function() {
  resetter.reset();
});
this.getRoot().add(resetButton, {left: 120, top: 10});
```

Form Object

We've already covered most parts of form handling. But one thing we've left out completely until now is layouting the form items. Thats where the `qx.ui.form.Form` widget comes into play.

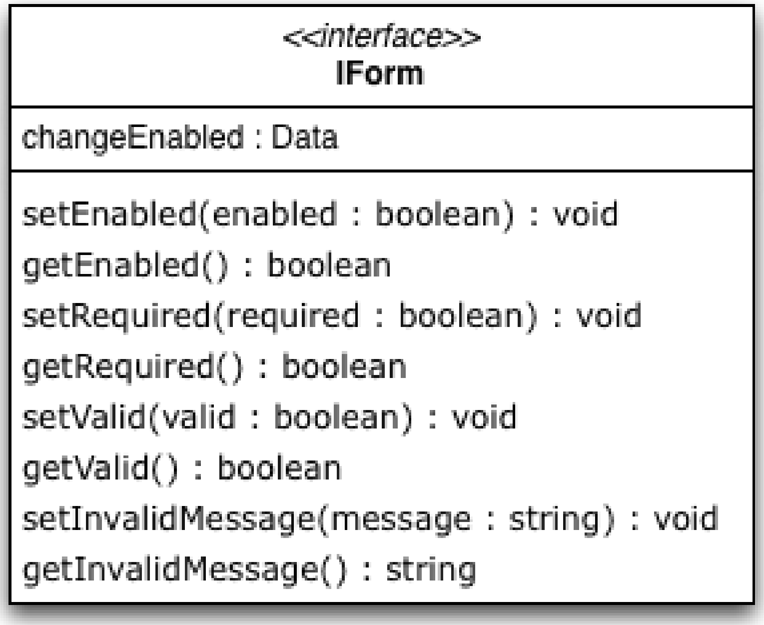
What is it?

The qooxdoo form is an object which includes three main parts.

- *Validation* using the `qx.ui.form.validation.Manager` class
- *Resetting* using the `qx.ui.form.Resetter` class
- Handling the layout of the form

As we have already talked about the first two items, I'll cover the last item in a more detailed way.

In most cases, a form's layout is specific to the application. It depends on the space available in the application and many other factors. Thats why qooxdoo has this flexible form layouting tool, which includes a set of default options to layout a form. On of the main requirements of the solution was extensibility so that anyone could have the layout their application requires. To get achieve this, we applied a pattern used widely across the qooxdoo framework, which moves all UI related code to renderer classes. These renderers are as lightweight as possible to make it easy for developers to write their own custom renderer, as you can see in this UML diagram:



Renderer

As the diagram shows, qooxdoo provides an interface for FormRenderer, the `IFormRenderer` interface. It defines two methods, one for adding a group of form items and one for adding buttons.

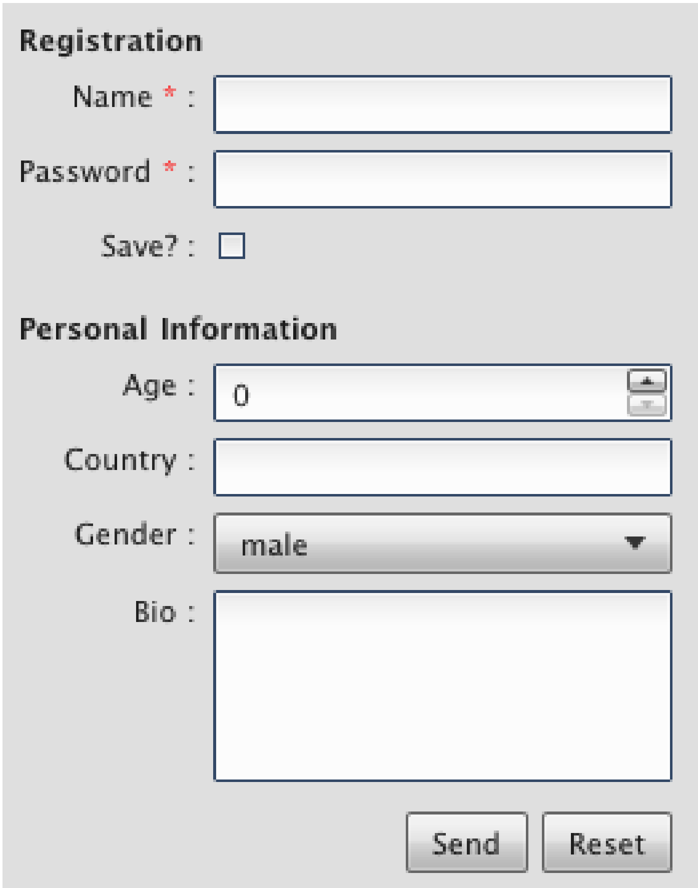
- addItems(items : qx.ui.form.IForm[], names : String[], title : String) : void
- addButton(button : qx.ui.form.Button) : void

Surely you've recognized the difference to the API of the form itself. Widgets are added to the form individually, but the renderer always gets a group of widgets at once. This gives the renderer additional information which it may need to render the form based on the number of groups rather then on the number of widgets.

You may ask yourself why we didn't use the layouts we usually use in such scenarios if we ant to render widgets on the screen. It may be necessary for a renderer to contain even more than one widget. Imagine a wizard or a form spread out over multiple tabs. That wouldn't be possible using layouts instead of renderer widgets.

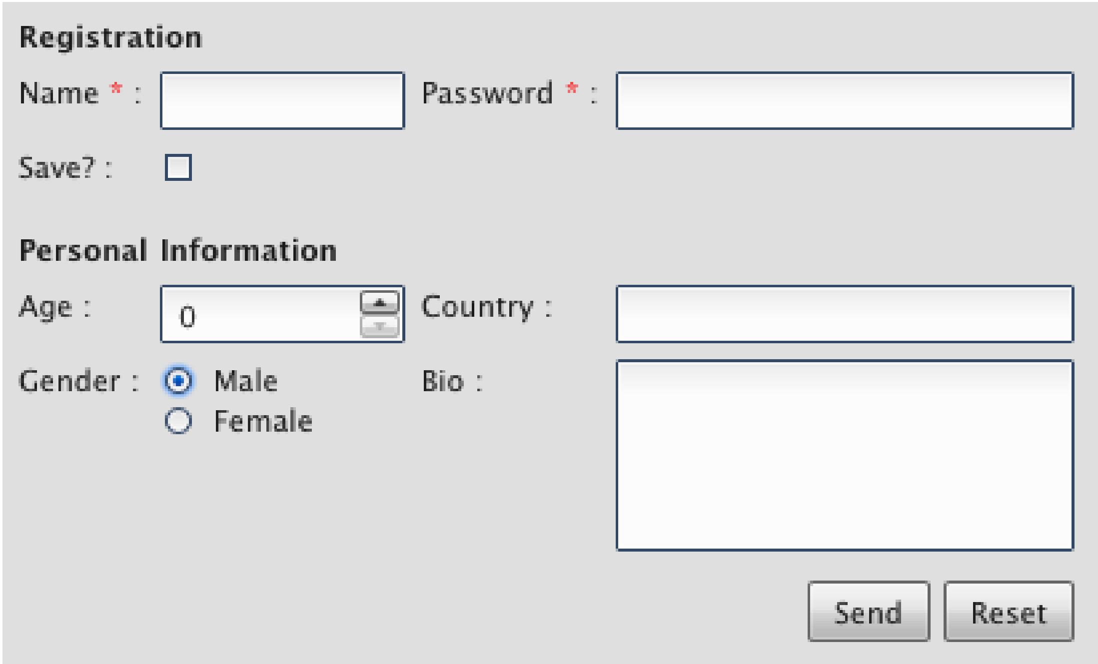
The following sections show the renderers included in qooxdoo, which can be used out of the box.

Default (Single Column) If you don't specify a renderer, the default is used, which is a single column renderer.

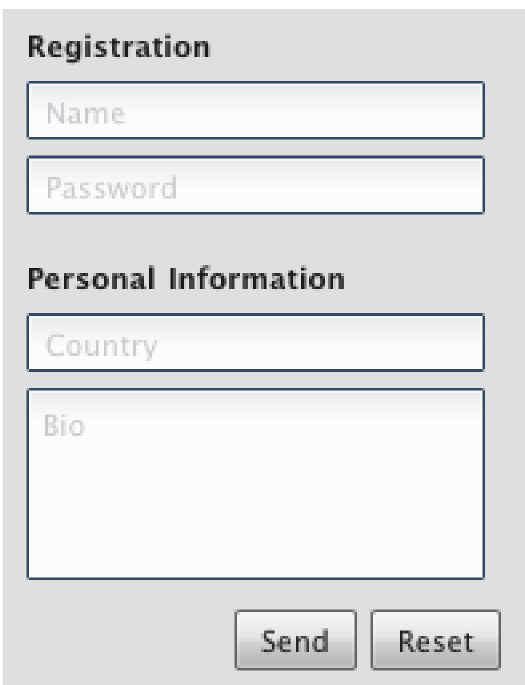


As you can see in the picture, the renderer adds an asterisk to every required field, adds a colon at the end of every label and defines the vertical layout.

Double Column The double column renderer has the same features as the previously introduced single column renderer but renders the fields in two columns, as you can see in the following picture.



Single Column with Placeholer This renderer is more a of demo showing how easy it can be to implement your own renderer. It has a limitation in that it can only render input fields which have the placeholder property. But the result is pretty nice:



Sample Usage

After we've seen how it should look, here come some examples showing how it works. In this example, we want to create a form for an address management tool. So we divide our input fields into two groups. The first group contains two text fields, one for the first name and one for the last name. The second group contains some contact data like email, phone number and company name. Finally, we want to add two buttons to the form, one for saving the data if it is valid and another for resetting the form. So here we go...

First, we need a form object.

```
// create the form
var form = new qx.ui.form.Form();
```

After that, we can create the first two input fields. As these two fields are required, we should mark them as such.

```
// create the first two input fields
var firstname = new qx.ui.form.TextField();
firstname.setRequired(true);
var lastname = new qx.ui.form.TextField();
lastname.setRequired(true);
```

As you can see, the input fields are text fields as described above. Next, we can add those input fields to the form.

```
// add the first group
form.addGroupHeader("Name");
form.add(firstname, "Firstname");
form.add(lastname, "Lastname");
```

First, we added a group header to create a headline above the two input fields. After that, we added them with a name but without a validator. The required flag we set earlier is enough. We need to add another group of input fields for the contact data.

```
// add the second group
form.addGroupHeader("Contact");
form.add(new qx.ui.form.TextField(), "Email", qx.util.Validate.email());
form.add(new qx.ui.form.TextField(), "Phone");
```

After adding the second group header, you'll see the text field for the email address, which uses a predefined email validator from the framework. The phone number does not get any validator at all. The last missing thing are the buttons. First, add the save button.

```
// add a save button
var savebutton = new qx.ui.form.Button("Save");
savebutton.addListener("execute", function() {
  if (form.validate()) {
    alert("You can save now...");
  }
});
form.addButton(savebutton);
```

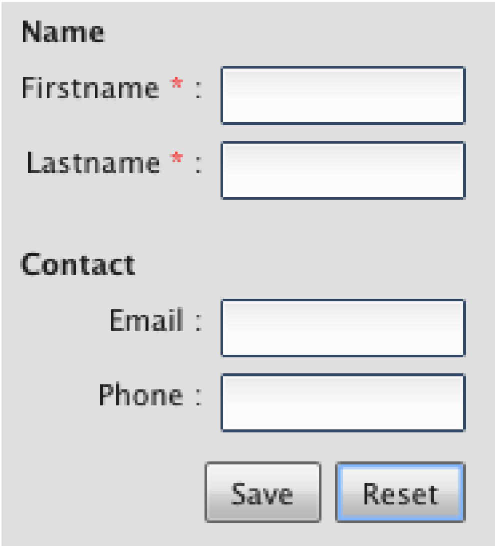
The save button gets an execute listener which first validates the form and, if the form is valid, alerts the user. The reset button is analogous.

```
// add a reset button
var resetbutton = new qx.ui.form.Button("Reset");
resetbutton.addListener("execute", function() {
  form.reset();
});
form.addButton(resetbutton);
```

Now the form is complete and we can use the default renderer to render the form and add it to the document.

```
// create the view and add it
this.getRoot().add(new qx.ui.form.renderer.Single(form), {left: 10, top: 10});
```

Running this code will create a form as described above which will look like this:



If you want to get a different look and feel, you can create a different renderer.

```
// create the view and add it
this.getRoot().add(
  new qx.ui.form.renderer.SinglePlaceholder(form),
  {left: 10, top: 10}
);
```

Just give it a try in the playground.

Form Controller

Data binding for a form certainly is a handy feature. Using a model to access data in the form brings form handling to another level of abstraction. That's exactly what the form controller offers.

The form controller is fully covered in the *data binding documentation*.

Sample Usage

The following example shows how to use the controller with a simple form, which contains three text fields: One for salutation, one for first name and one for last name.

First, we create the form:

```
// create the form
var form = new qx.ui.form.Form();
```

In a second step we add the three text fields. The important thing here is that if no name is given - as in the first two cases - each label will also be used as a name. For that, all spaces in the label are removed.

```
// add the first TextField ("Salutation" will be the property name)
form.add(new qx.ui.form.TextField(), "Salutation");
// add the second TextField ("FirstName" will be the property name)
form.add(new qx.ui.form.TextField(), "First Name");
// add the third TextField ("last" will be the property name)
form.add(new qx.ui.form.TextField(), "Last Name", null, "last");
```

After we add the text fields, we can add the view to the application root.

```
// add the form to the root
this.getRoot().add(new qx.ui.form.renderer.Single(form));
```

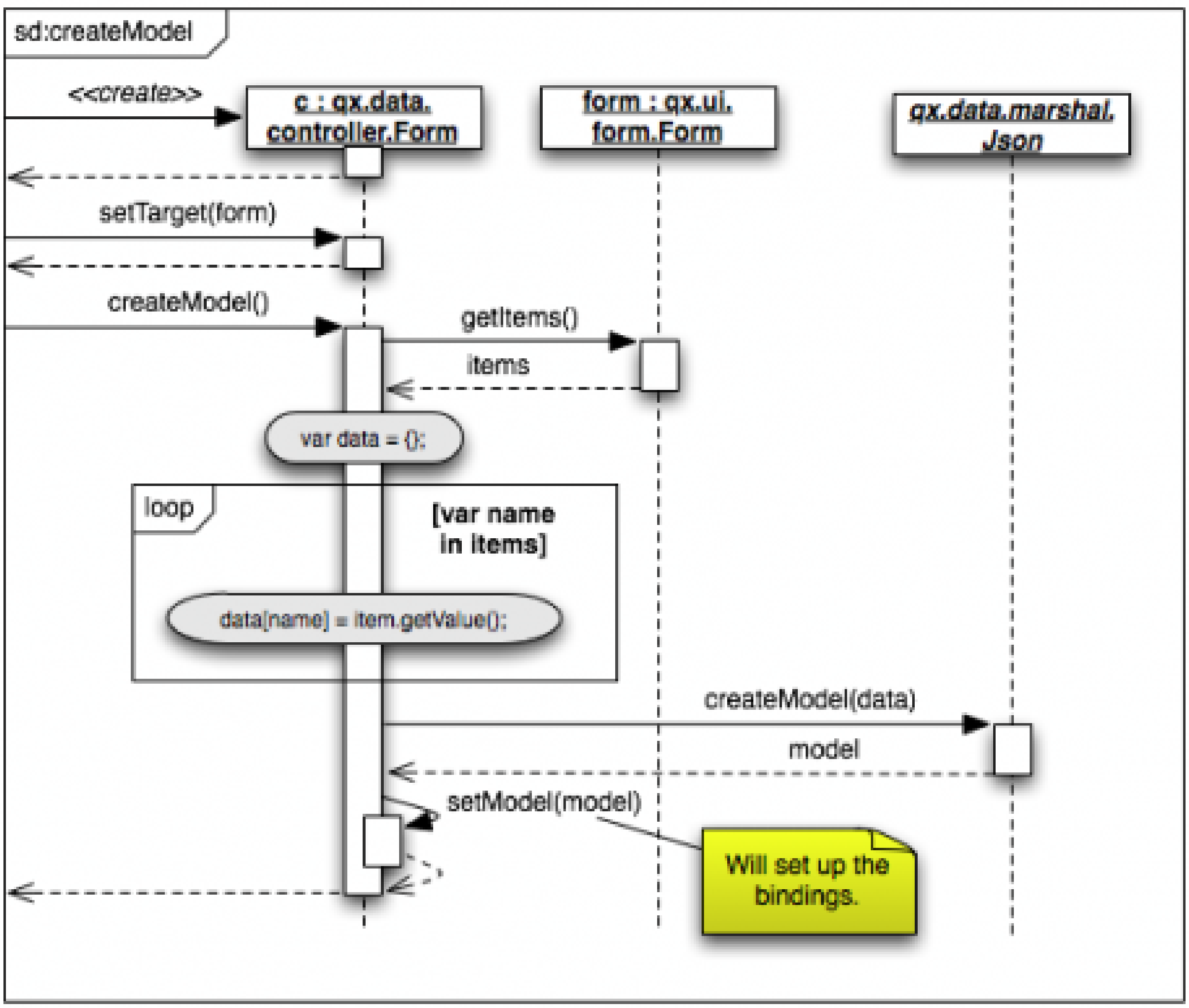
Now that the form has been created, we can take care of the data binding controller. We simply supply the form instance as an argument to the constructor. But we don't have a model yet, so we just pass `null` for the model.

```
// create the controller with the form
var controller = new qx.data.controller.Form(null, form);
```

The final step for data binding is to create the actual model.

```
// create the model
var model = controller.createModel();
```

Take a look at the following sequence diagram to see how it works internally.



Now we have managed to set up a form and a model connected by bidirectional bindings. So we can simply use the model to set values in the form.

```
// set some values in the form
model.setSalutation("Mr.");
```

```
model.setFirstName("Martin");
model.setLast("Wittemann");
```

As you can see here, the properties (and therefore setters) are defined according to the names we gave the text fields when adding them.

See the code in action in the playground.

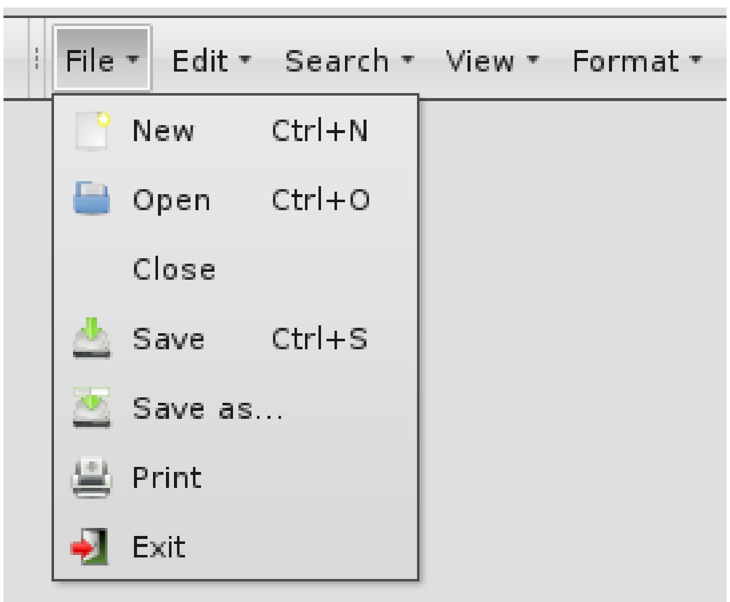
Still to come...

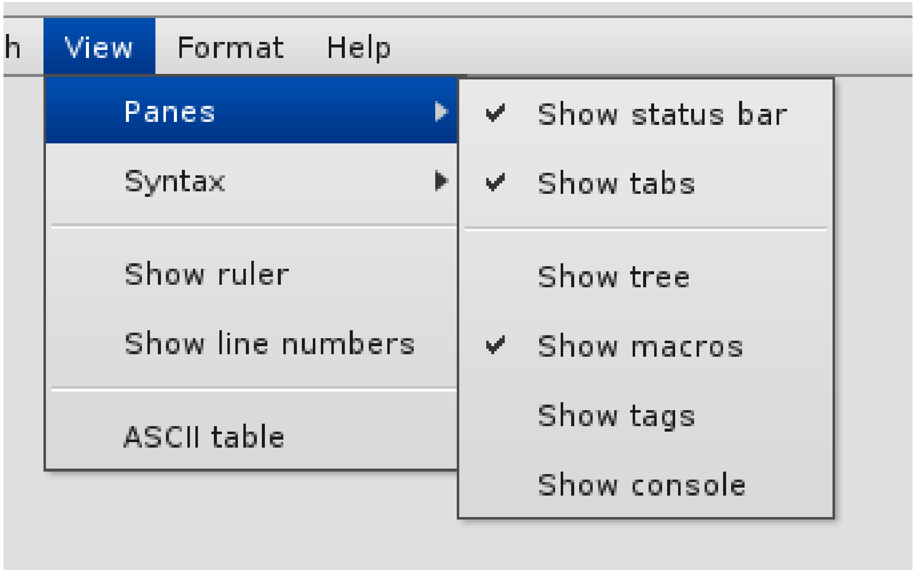
- A way to create a form out from a JSON definition

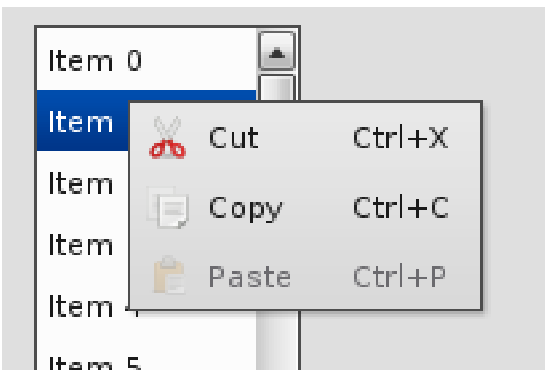
4.2.10 Menu Handling

Menus are well-established user interface elements in GUIs. They are popup-like controls that provide simple or cascading lists of buttons. Typical uses show menus opening off from buttons in tool bars, or popping up as context menus on mouse right-clicks e.g. on a tree element.

Here are a few examples:







The Demobrowser provides further examples.

Menus can be constructed in a qooxdoo application using widgets from the qx.ui.menu name space. The main class from this package is `Menu`. Other classes allow you to tailor the appearance and the behaviour of the menu you create. You can even use checkboxes and radiobuttons inside your menus.

Simple Example

Here is a simple menu example:

```
// Create the menu
var menu = new qx.ui.menu.Menu();

// Creates the command
var command = new qx.event.Command("Control+O");
command.addListener("execute", function() {
  this.debug("Open action");
},this);

// Add some content
var openButton = new qx.ui.menu.Button("Open", "icon/16/actions/document-open.png", command);
var closeButton = new qx.ui.menu.Button("Close");

menu.add(openButton);
menu.add(closeButton);
```

```
// Add behaviour
closeButton.addListener("execute", function() {
  this.debug("Close action");
}, this);

// Create a button that will hold the menu
var button = new qx.ui.form.MenuButton("Menu", null, menu);
```

There are a couple of things to note here:

- The main widget is the `menu` of type `qx.ui.menu.Menu`.
- Menu buttons are of type `qx.ui.menu.Button` and are created individually.
- They are then added to the menu. The buttons will appear in the menu in the order they are added.
- The `closeButton` is created with the minimal set of parameters, namely just the string for the button label. For a more advanced solution, see the `openButton`: you can optionally specify a button icon, and a command `qx.event.Command` that is invoked if the button or the shortcut is pressed/selected.
- You can supply missing or updated features after the widget's creation; e.g. the callback function for the `closeButton` is provided in a separate method call to `addListener()`.
- The canonical event for the selection of a menu button is the `execute` event. (This is in line with other button flavors throughout the qooxdoo framework, e.g. the regular `qx.ui.form.Button`).

Complex Menu Sample

This example should show how to create a menu structure with submenu and how to handle with groups.

Qooxdoo has some widgets that need a menu to handle user interaction. For this sample we will chose the `qx.ui.toolbar.ToolBar` to create the menu structure. To see a overview, witch widgets uses a menu, take a look in the *Menu*.

This code snippet show how to create a "ToolBar" with to menu items "File" and "View":

```
// Create the toolbar and add to the DOM
var toolBar = new qx.ui.toolbar.ToolBar();
this.getRoot().add(toolBar, {
  left: 20,
  top: 20,
  right: 20
});

// Create "File" menu
var fileButton = new qx.ui.toolbar.MenuButton("File");
toolBar.add(fileButton);

var fileMenu = new qx.ui.menu.Menu();
fileMenu.add(new qx.ui.menu.Button("New", null, null, this.__getNewMenu()));
fileMenu.add(new qx.ui.menu.Button("Open...", "icon/16/actions/document-open.png"));
fileMenu.add(new qx.ui.menu.Separator());
fileMenu.add(new qx.ui.menu.Button("Save", "icon/16/actions/document-save.png"));
fileMenu.add(new qx.ui.menu.Button("Save As...", "icon/16/actions/document-save-as.png"));
fileMenu.add(new qx.ui.menu.Separator());
fileMenu.add(new qx.ui.menu.Button("Exit", "icon/16/actions/application-exit.png"));
fileButton.setMenu(fileMenu);

// Create "View" menu
```

```
var viewButton = new qx.ui.toolbar.MenuButton("View");
toolBar.add(viewButton);
var viewMenu = new qx.ui.menu.Menu();
viewMenu.add(new qx.ui.menu.Button("Panes", null, null, this.__getPanesMenu()));
viewMenu.add(new qx.ui.menu.Button("Syntax", null, null, this.__getSyntaxMenu()));
viewMenu.add(new qx.ui.menu.Separator()); // First kind to add a separator
viewMenu.add(new qx.ui.menu.CheckBox("Show ruler"));
viewMenu.add(new qx.ui.menu.CheckBox("Show line numbers"));
viewMenu.addSeparator(); // A other kind to add a separator
viewMenu.add(new qx.ui.menu.Button("ASCII table..."));
viewButton.setMenu(viewMenu);
```

There are a couple of things to note here:

- The `qx.ui.menu.Menu` could get some different children (Button, Seperator, CheckBox, ...)
- The fourth parameter in `qx.ui.menu.Button` is also a menu. So it is possible to create submenus.
- There are tow kinds to add a separator to a menu. The first kind is to create a `Separator` instance and add this to the menu. Or the other kind is to call the `addSeparator` method from the `Menu` instance.

The next code snipped should explain how to create a menu, which contain RadioButtons, but only one could be selected:

```
__getSyntaxMenu : function()
{
  var syntaxMenu = new qx.ui.menu.Menu();

  var cDialectMenu = new qx.ui.menu.Menu();
  cDialectMenu.add(new qx.ui.menu.RadioButton("C"));
  cDialectMenu.add(new qx.ui.menu.RadioButton("C Sharp"));
  cDialectMenu.add(new qx.ui.menu.RadioButton("C Plus Plus"));

  var htmlButton = new qx.ui.menu.RadioButton("HTML");
  var jsButton = new qx.ui.menu.RadioButton("JavaScript");
  var cdialectButton = new qx.ui.menu.Button("C Dialect", null, null, cDialectMenu);
  var pythonButton = new qx.ui.menu.RadioButton("Python");

  syntaxMenu.add(htmlButton);
  syntaxMenu.add(jsButton);
  syntaxMenu.add(cdialectButton);
  syntaxMenu.add(pythonButton);

  // Configure and fill radio group
  var langGroup = new qx.ui.form.RadioGroup();
  langGroup.add(htmlButton, jsButton, pythonButton);
  langGroup.add.apply(langGroup, cdialectButton.getMenu().getChildren());

  return syntaxMenu;
}
```

You can see, that the menu contains `RadioButton` and all `RadioButton` should grouped in one `RadioGroup`, but the `RadioButton` in the submenu "C Dialect" should also be considered in the `RadioGroup`.

To add a `RadioButton` to the `RadioGroup` call the `add()` method from the `RadioGroup`. The parameter from `add()` is a variable number of items which should be added. You can see that the code calls a `langGroup.add.apply()` method to add the `RadioButton` from the "C Dialect" submenu. This is no qooxdoo construction, the `apply()` method is a construction from JavaScript and it is not important to know how thus the method works.

Additional Menu Topics

Menu positioning

Qooxdoo will go a long way to position a menu sensibly and with regard to the enclosing container, so that menu buttons are always fully visible if the menu is opened.

The Placement demo shows how the menus are positioned.

4.2.11 Window Management

Window is a widget used to show dialogs or to realize a MDI (Multiple Document Interface) applications. Windows can only be added to `qx.ui.window.Desktop` widgets, or to widgets which implement the `qx.ui.window.IDesktop` interface.

Each Desktop widget must have a `qx.ui.window.Manager`. If none is provided, the default window manager (`qx.ui.window.Window#DEFAULT_MANAGER_CLASS`) is used. The desktop uses the manager to handle the contained windows.

The manager takes care of windows z-index order. Windows can be normal (default), always on top or modal. Always on top windows stay on top of normal windows and modal windows appear in front of all other windows. If there are a bunch of windows open and we close one, the manager will activate the window that is higher in the z-index order stack.

Let's see this in action. We'll create a tabview with one page, create a desktop widget for the page, and add different types of windows. You can see how the opened windows stack on each other and when you close one, the highest z-index order window will get activated.

```
var root = this.getRoot();
var tabView = new qx.ui.tabview.TabView();
var page = new qx.ui.tabview.Page("Desktop");
var windowManager = new qx.ui.window.Manager();
var desktop = new qx.ui.window.Desktop(windowManager);
var aWindow = null;

page.setLayout(new qx.ui.layout.Grow());
page.add(desktop);
tabView.add(page);
root.add(tabView, {edge: 0});

//create 3 normal windows and add them to the page's desktop
for (var i=0; i<3; i++)
{
  aWindow = new qx.ui.window.Window("Normal Window #" +i).set({
    width:300
  });
  desktop.add(aWindow);
  aWindow.open();
}

//create 3 alwaysOnTop windows and add them to the page's desktop
for (var i=0; i<3; i++)
{
  aWindow = new qx.ui.window.Window("AlwaysOnTop Window #" +i).set({
    width:300
  });
  aWindow.setAlwaysOnTop(true);
```

```
  desktop.add(aWindow);
  aWindow.open();
}

//create a modal window and add it to the page's desktop
aWindow = new qx.ui.window.Window("Modal Window #" +i).set({
  width:300
});
aWindow.setModal(true);
desktop.add(aWindow);
aWindow.open();
```

Like I said, windows can be added to widgets which implement the IDesktop interface. This interface is implemented by `qx.ui.window.MDesktop` mixin. You can use this mixin to make a custom widget behave like a Desktop. This is exactly what the superclass of all root widgets (`qx.ui.root.Abstract`) does. This is why we can add windows to a root widget.

```
var win = new qx.ui.window.Window("First Window");
var root = this.getRoot();
root.add(win);
win.open();
```

Related documentation

Window widget

Demos and API

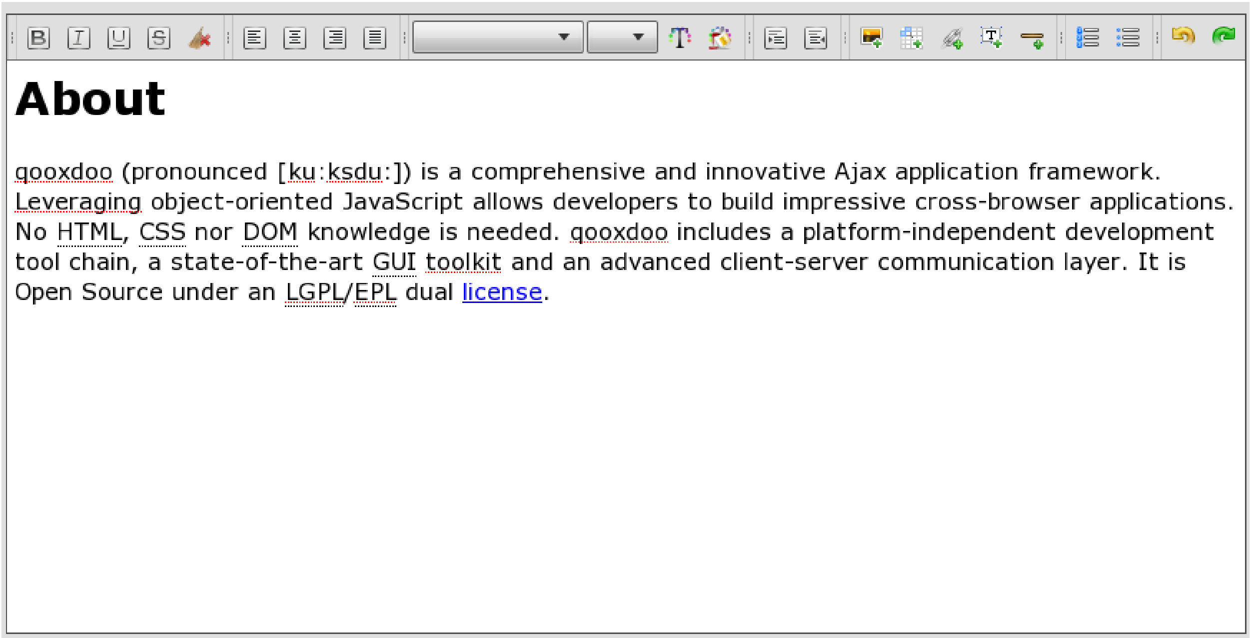
To find out more, you can check the desktop demo and the API reference.

4.2.12 HTML Editing

HtmlArea is a html editing widget which is part of the framework. This widget is available as low-level and UI-level implementation. The first targets traditional webpages / single-page applications and the latter Rich Internet Applications (RIA) as preferred usecase.

Here you can find some interesting technical info.

Demo



Setup

One important step is necessary to get the HtmlArea up and running.

Note: If you setup the component without handing the **source** parameter you have to place a **blank.html** file next to your applications **index.html**.

This is necessary due the Same-Origin Policy implemented by most browsers.

Features

Feature List

This page aims to describe the features of the HtmlArea component. *Aims* because there are for sure features which are missing or considered as *must-have* to not enter the feature list as own entry.

This page should get you a good overview of what you can expect from this HTML editing component.

End-User Features

Text Formatting

- Bold
- Italic
- Underline
- Strikethrough
- Text Color

- Background Color
- Font Size
- Font Family

Alignment

- Left
- Center
- Right
- Justify

Lists

- Unordered lists
- Ordered lists

Inserting

- Tables
- Images
- Horizontal rulers
- Hyperlink
- HTML code

Document Wide Formatting

- Background Image
- Background Color

Additional Features

- Removing format
- Select the whole content
- Indent / Outdent
- Undo / Redo

Developer Features

Events

- Load / LoadingError and Ready
- Current cursor context
- Contextmenu
- Focus / Focus out

Content Manipulation

- Content as HTML output
- Post-process HTML output
- Current selected HTML
- Reset content
- Context Information of current focused node (e.g. to update a toolbar widget)

Advanced Paragraph-Handling

- Keeps formatting across multiple paragraphs
- Type of line-break adjustable (new paragraph or new line)
- Support for `Shift+Enter` and `Ctrl+Enter` to insert single line-break

Additional Features

- Hotkey Support
- Set own CSS for content at startup
- Access to content document and content body
- Access to editable iframe element

Technical Feature List

In comparison to the *Feature List* of the HtmlArea this page describes some technical insights of the component. If you plan to get to know some details of how to develop a WYSIWYG component and want to learn the pitfalls of the different browser implementations this is good place to start.

Startup The HtmlArea relies on a editable iframe. To take control over this iframe the component has to ensure that the iframe's document is fully loaded and accessible. For every browser the `load` event of the iframe object is used. Only for IE it is necessary to poll the document if it's not immediately available after the `load` event. The result of the startup phase is the `ready` event which informs the application developer that the startup was successful.

Content Wrapping Since the application developer only sets the content of the HtmlArea and not the whole document the component needs to setup the rest of the content (`DOCTYPE`, `HTML` and `BODY` elements). The difficult exercise here is to set the right style attribute at the right element for each browser.

The toughest thing is to get the right behaviour for native scrollbars. In IE for example the overflow handling with `overflow-x` and `overflow-y` does not work correctly. When both style attributes are set IE does mix them up by overwriting one with the others value.

Anyway, the correct content wrap is important for

- document is taking the whole space of the iframe
- no margins and paddings are set
- scrollbars are only shown if the user enters more content than space is available

Editable Document Another pitfall is how to set the document of the iframe object editable. There are two properties which can be be applied for an editable document: **designMode** and **editable**.

The `designMode` property is applied for all browsers and works at the `document` node of the iframe.

Setting the `editable` property is only needed for gecko browsers. And only if the HtmlArea was hidden and shown again. The `editable` property is applied to the `body` element.

Internet Explorer For IE it is important to set the document design mode **before** the content is rendered. Once the document is editable it does not loose this status even if the whole component is hidden and shown again.

Gecko, Webkit and Opera All three need to have rendered content to set the document design mode correctly.

Focus Management At least IE has problems whenever a native command (*execCommand* method) does manipulate the content of the editable iframe and the iframe document does **not** have the focus. If an application developer want to use a toolbar to offer the user an interface to manipulate the content he has to make sure that each of these buttons need a special setup. Otherwise the button would *steal* the focus from the editing component whenever clicked.

Luckily qooxdoo does offer this customization out-of-the-box. The application developer only has to set the properties *keepFocus* to *true* and *focusable* to *false*.

```
button.set({
  focusable: false,
  keepFocus: true
});
```

Advanced Key Events One major feature is to track the user input. To use the powerful key event handler in qooxdoo the HtmlArea does listen to all key events at the `body` element and handles various actions depending on the user's input. This way it is possible to work with a `keyIdentifier` instead of the `keyCode` or `charCode`.

Integration Guide

Integrate the HtmlArea into your application

Note: This explanations mainly do address the `0.8+ based` HtmlArea component.

This page does explain what you should consider when integrating the HtmlArea component in your application. However, it does **not** explain how to setup the component itself, it's rather an integration guide to avoid pitfalls.

Use Public API This one should be self-explaining. Do not use any internal API to get things done even it's the easy way to go. If it's hidden from the application developer then by purpose, but if you need access to specific parts of the component which is not offered don't hesitate to file a bug report.

Use Events The component does offer various events to work with e.g. the `ready` event to get informed of the finished loading.

The bottom line is the same as for the public API: use these events to interact with the component. If an event is missing feel free to file a bug report.

Lazy Initialization The HtmlArea widget is using a low-level editing component to offer a WYSIWYG editor solution. The widget does initialize this editing component after the first appear of the widget. So if you use e.g. a stack container which hides the HtmlArea keep in mind that the widget is only fully usable after it is shown.

Toolbar Details The HtmlArea does only offer the plain editing widget so if you do not use the HtmlEditor contribution and instead create your own toolbar you have to consider some specialities concerning the focus management of qooxdoo.

Since the HtmlArea relies on that the focus is not lost to another widget (e.g. a toolbar button) during the execution of a command you have to set two focus-specific properties on each widget which runs commands at the HtmlArea component.

The two properties `keepFocus` and `focusable` have to be used together to get the correct behaviour. The more important property is `keepFocus` which certainly ensures that the given widget `never` get the focus - even if this widget is clicked. This will leave the focus at the HtmlArea component solving many focus-related issues successfully (especially for IE browsers).

Example code snippet

```
button = new qx.ui.toolbar.Button(null, iconURL);
button.set({ focusable : false, keepFocus : true });
```

No Own Focus Management As already mentioned the focus management is important for HTML editing widgets and there are special solutions necessary for the component already. Implementing an own focus management on top in your application code `can` cause problems for your users. So if you encounter any issues that the component e.g. does not perform a certain command even a button is clicked it's probably a focus-related issue. As always: the component is far from perfect, don't hesitate to file a bug report for issues you encounter.

Keyboard Shortcuts Since you can use *keyboard shortcuts* to manipulate the content you should not implement shortcuts with the same key bindings. A possibility to disable the shortcuts completely will soon be available. See Bug #1193 for details.

Available Keyboard Shortcuts

The result of using the shortcuts `Control + Enter` and `Shift + Enter` are explained at the *Paragraph Handling* page.

The following keyboard shortcuts are implemented at the moment:

- `Ctrl + A` - Select the whole content
- `Ctrl + B` - Toggle the current selection to Bold / Normal text
- `Ctrl + I` and `Ctrl + K` - Toggle the current selection to Italic / Normal text
- `Ctrl + U` - Toggle the current selection to Underline / Normal text

If the Undo / Redo functionality is enabled the following shortcuts are additionally available:

- `Ctrl + Z` - Undo the last change
- `Ctrl + Y` - Redo the last undo step
- `Ctrl + Shift + Z` - Redo the last undo step

Recommendations

This page should help developers using the HtmlArea to stick with some recommendations to avoid known issues or to call attention how to use a specific feature.

Common Font Families Since the HtmlArea *"only"* is a editing component it does not offer a complete toolbar or other features which an full-blown Html Editor might offer. So if you setup an own toolbar and decide to offer the user a possibility to change the default font family you should be careful not to use a font family which is not widely available. If the client computer does not has the listed font family installed it will certainly fall back to the systems default. The user will be irritated by different choices which end up with the same result if he applies them to his written content.

To avoid this problem you should play safe and offer the following font families:

- Arial
- Arial Black
- Verdana
- Courier New
- Courier
- Georgia
- Impact
- Comic Sans MS
- Tahoma
- Lucida Console

A nice list of the most common font families is listed at CodeStyle.org

InsertHtml Command This command lets you insert you HTML code directly into the component's document. It is powerful and can be an easy way to accomplish your goals, but you should keep in mind that this method should only be used if there is no other possibility offered.

If you e.g. want to insert an image into the document use the dedicated `insertImage` command instead of putting your HTML code together.

Avoid DIV elements with fixed width or height The problem with DIV elements which have *width* or *height* set with CSS styles is that IE offers for those DIV elements resize/move handles. This is in the most cases not desired. So better use *margin*, *padding* or *top|left|right|bottom* to position your DIV element.

Additionally if you set a width of 600px to a DIV element users with a small resolution (like 800 x 600) might end up with horizontal scrollbars.

Technical Background

HTML Editing In General

External Information

General infos

- Rich HTML editing - Part 1

Browsers

Mozilla ("Midas")

- Midas specification
- Demo
- Migrationguide IE -> Gecko
- Documentation
- Source code (see list under MidasCommand in nsHTMLDocument.cpp)
- DOM Client Object Cross-Reference

IE ("HTML Edit")

- MSDN Overview and tutorials
- Documentation
- Overview of Command Identifiers
- A Note about the DHTML Editing Control in IE7+

Opera

- Opera Browser Wiki

Safari

- WebKit: HTML Editing
- Quietly, Safari Finally Gains WYSIWYG Editing Powers
- execCommand list
- WYSIWYG comes to Safari 1.3

Compatibility

- The Mozile project contains code which adapts Internet Explorer's Selection object to an interface like Mozilla's.
- Converting your app from IE to Midas
- execCommand compatibility

General

- htmlarea.com
- cmsreview.com
- geniisoft.com
- Web-Based Rich Text Editors Compared

Overview of exisiting WYSIWYG editors

Editor	License	Pro/Con
YUI RTE	BSD	**Pro**: works with all well-known browsers (IE / Gecko / Ope **Con**: Still in Beta (although the final release version should
Xinha	HTMLArea (BSD based)	
RTE	Creative Commons	
RTEF	MIT	**Pro**: works with all well-known browsers (IE / Gecko / Ope **Con**: no user-feedback e.g. which font or size is currently us
WYMEditor	MIT/GPL	**Pro**: produces XHTML, uses CSS; **Con**: currently only avai
dojo	BSD	
TinyMCE	LGPL	
FCKEdit	GPL, LGPL and MPL	
Solmetra	GPL	
FreeRTE	Creative Commons	
CMSimple	AGPL	
XStandard lite	Freeware	
Loki Whizzywig	GPL	

command	Mozilla	IE	Opera	Safari
Bold	x	x	x	x
Italic	x	x	x	x
Underline	x	x	x	x
Strikethrough	x	x	x	x
Color				
BackColor	x	x	x	x
ForeColor	x	x	x	x
HiliteColor	x			x
Font Handling				
FontName	x	x	x	x
FontSize	x	x	x	x
IncreaseFontSize	x		x	
DecreaseFontSize	x		x	
Subscript	x	x	x	x
Superscript	x	x	x	x
Formatting and CSS				
ContentReadOnly	x		x	
StyleWidthCSS	x			
UseCSS	x		x	
RemoveFormat	x	x	x	x

Continued on next page

Table 4.1 – continued from previous page

User actions				
Copy	x	x		x
Paste	x	x		x
Cut	x	x	x	x
Delete	x	x	x	x
Undo		x	x	x
Redo		x	x	x
Print		x		x
SaveAs		x		
Alignment				
JustifyLeft	x	x	x	x
JustifyCenter	x	x	x	x
JustifyRight	x	x	x	x
JustifyFull	x		x	x
Indent	x	x		x
Outdent	x	x		x
Hyperlinks				
CreateLink	x	x	x	x
Unlink	x	x	x	x
Lists				
InsertOrderedList	x	x	x	x
InsertUnorderedList	x	x	x	x
Basic (formatting) elements				
FormatBlock	x	x	x	x
Heading	x			
InsertParagraph	x	x	x	x
InsertImage	x	x	x	x
InsertButton		x		
InsertFieldset		x		
InsertHorizontalRule		x	x	x
InsertHTML	x		x	x
InsertIFrame		x		
Form elements				
InsertInputButton		x		
InsertInputCheckbox		x		
InsertInputFileUpload		x		
InsertInputHidden		x		
InsertInputImage		x		
InsertInputPassword		x		
InsertInputRadio		x		
InsertInputReset		x		
InsertInputSubmit		x		
InsertInputText		x		
InsertSelectDropdown		x		
InsertSelectListbox		x		
InsertTextArea		x		

Continued on next page

Table 4.1 – continued from previous page

InsertMarquee		x		
Bookmarking				
CreateBookmark		x		
UnBookmark		x		
Selection and status handling				
SelectAll	x	x	x	x
Unselect		x	x	x
MultipleSelection		x		
Overwrite		x		
Refresh		x		
Misc				
2D-Position		x		
AbsolutePosition		x		
LiveResize		x		
gethtml	x			
contentReadOnly	x			
insertBrOnReturn	x			
enableObjectResizing	x			
enableInlineTableEditing	x			

Browser-specific overview of "execCommand"

Copy and Paste

For a HTML editor component it is important to get along with external content which is inserted with a `Copy and Paste` operation. This is especially important if any filter for the external content should be applied before the content is actually inserted in the editor.

However it is quite difficult to implement this across all major browsers. This short article should give a short overview about the existing events in the different browsers.

To get the detailed overview on this topic check out the section at quirksmode.org

IE This browser offers the most events. Besides `onpaste` and `oncopy` there are also events like `beforepaste`, `beforecopy` and `beforecut`. Additionally all events are stoppable and are bubbling up the DOM hierarchy.

Safari Follows almost the implementation of IE and goes partly beyond it. Safari offers a wide range of events to detect a `Copy and Paste` operation, but has currently no implementation at image elements.

Gecko In Firefox 2 there is no support for any event to detect a `Copy and Paste` operation directly. One can detect the pressed shortcuts, but if the user paste some text via the menu/contextmenu there is possibility to catch that. With the upcoming release of Firefox 3 this situation will improve. This version will have some support for such events like onpaste or oncopy

Opera Same situation as Firefox 2: no working implementation for `copy` or `paste` events.

Text align

The text align of a selction can be modified using the following exec commands: `JustifyLeft`, `JustifyCenter`, `JustifyRight` and `JustifyFull`.

Browsers

- **IE**: Text align is applied on the paragraph which contains the selection.
- **Gecko** and **Opera**: Text align is applied on selection only. The selection gets surrounded by a `<div>` tag containing a `text-align` style attribute.
- **Webkit**: Applies `text-align` style attribute on every `<div>` element that is (partly) selected.

Problems

- If `<br>` tags are used for line breaks, the textalign will be applied on the `<p>` tag in IE, even if only a part of this `<p>` has been selected!
- If `<p>` tags are used for line breaks, all style settings set will be "lost" after entering an other `<p>` tag in FF. It is necessary to "save" these settings manually and apply them on the new paragraph.

Browser Bugs

Gecko

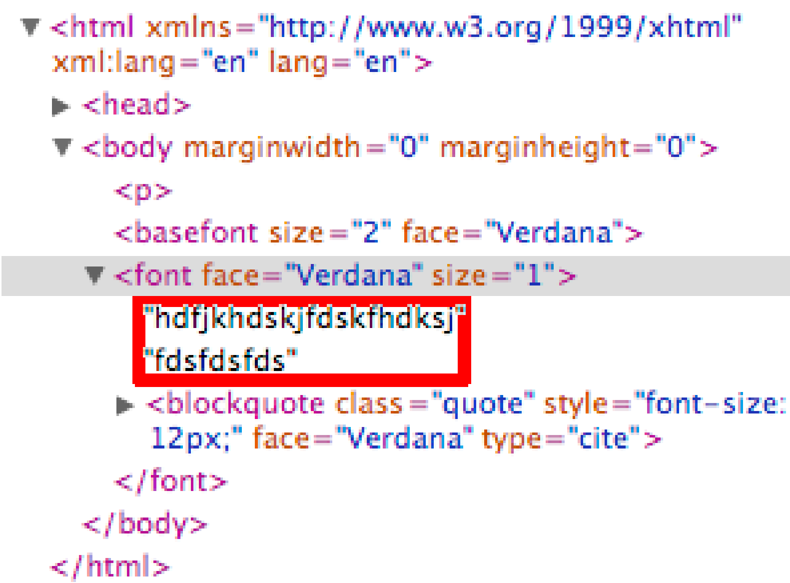
- **Gecko 1.8** *needs* a `<br>` tag inside an element with `contenteditable="true"`, even if the element is empty! If no such element existes, Gecko automatically adds it. These elements can be recognized by the proprietary attribute *_moz_editor_bogus_node*: `<br _moz_editor_bogus_node="TRUE" _moz_dirty=""/>`
- **Gecko 1.9** will *always* insert this `<br>` tag, if `contenteditable="true"` is set. Even if the element contains content! This `<br>` tag is removed, as soon as any input is entered by the user: https://bugzilla.mozilla.org/attachment.cgi?id=119342
- **Undo/Redo** : it *could* happen that 2 content changes occuring right after another leading Gecko to remove both of these 2 changes in **one** undo step. This is especially important for the undo/redo stacks of the HtmlArea.

Internet Explorer

- If you want to use the `pasteHTML()` function, you have to select the textrange first using `select()`.

Webkit/Safari

- Setting a background color for text on *collapsed* selection is not working like in Gecko or IE. Instead of setting the background color and allowing the user to type ahead in the new background color (like in Gecko/IE) nothing happens. The current solution in the HtmlArea is to select the word currently under the caret and to set the background color on this selection. Working on a user-selection works as expected.
- Deleting a block element (e.g. an <p> tag) can cause an element to contain *two* text nodes:



This wrong behavior can cause problems with selections.

Default Paragraph Handling

This section describes how browsers and other applications react on different keys to enter line breaks or paragraphs.

P = paragraph (`<p>` tag)

LB = line break (`<br />` tag)

Browsers

	Firefox	MSIE	Opera	Webkit
<enter>	LB	P	LB	<div>
<shift> + <enter>	LB	LB	LB	<div>
<strg> + <enter>	—	—	—	—

Word processors / E-mail clients

	MS Word	OO Writer	Outlook	Thunderbird
<enter>	P	P	P	LB
<shift> + <enter>	LB	LB	LB	LB
<strg> + <enter>	Page break	P	—	LB

Implementation Details

Undo and Redo

Limitations The implementation of undo/redo in the HtmlArea has some limitations you should be aware of. It is possible to `undo` all of your steps but `redo` is only possible when no other action occured between the undo and the redo action. If you undo several steps and e.g. enter some text you **can not** execute redo anymore.

Note: If you use the Undo/Redo functionality you have to make sure you are not manipulating the content of the HtmlArea by using the `innerHTML` property of an element.

This will break Undo/Redo functionality!

Implementation: Description on a high-level The implementation is split up into two different approaches.

For Internet Explorer the `execCommand` approach can't be used anymore. The internal undo / redo stack gets broken on every DOM manipulation. So, if any qooxdoo decorator is used this approach is a dead end. Instead an own implementation using `innerHTML` is used for IE browsers.

For all other browsers the base of the Undo/Redo functionality is to use the `execCommand` method to manipulate the content **whenever** possible. Each change which is performed with a call of `execCommand` is easy to undo/redo. For any manipulation which cannot be achieved using the built-in `execCommand` a special implementation for each browser is necessary (e.g. changing the background color of the whole document).

Using the Decorator Pattern To easily integrate the undo/redo management with the commands of the HtmlArea the `UndoManager` class is a decorator of the `CommandManager` class. It takes the method calls from the HtmlArea class, collects the info for undo the action and calls the decorated `commandManager` class to actually perform the requested action. This keeps both implementations clean and separated.

Tracking changes using stacks Two stacks keep track of the changes which are done to the content: an **undo stack** and the corresponding **redo stack**. Currently each stack holds four different types of changes:

- Command
- Content-block
- Custom
- Internal

Each entry in the stacks is represented by an object which holds additional info (the type above is among this info).

Command Every change which is performed with the `execCommand` method is equipped with this type. These changes are the easiest to track and to undo/redo.

Content-block Each keypress event is observed to determine changes in the content and to mark a set of content changes as an own block which is capable for an undo/redo step. For example IE and Gecko do both recognize text changes as a content block if the text changes occured between two calls of `execCommand`.

Custom These changes are the ones which cannot be handled with the built-in `execCommand` method. For example changing the background color of the whole document is a custom undo/redo step which needs to be handled in a special way by each browser.

Internal These steps are included to keep the stacks intact if the user e.g. resizes an images with the handles provided by the browsers. It is possible to undo/redo these internal changes with the common `execCommand` method. The primary task here is to record these changes and add them to the stack(s).

Paragraph Handling

The aim of the component is to facade all the browser differences concerning the behaviour when the user hits the `Enter`, the `Shift+Enter` or the `Control+Enter` combination. And this is by far not an easy task since the differences between the browsers are enormous.

Formatting across multiple paragraphs Every formatting infos like *underline*, *bold*, *text color*, *text size* etc. are transferred to the new paragraph. It is likely that the user expects to write on with the same configuration/modifications he applied to the former paragraph.

Alignment A paragraph in always aligned completely - the way a word processor also work. This *can* be irritating at the first time of use if e.g. a paragraph contains multiple lines of text each separated by normal line-breaks, but concerning alignment the paragraph is treated only as whole. So every line of the paragraph (=the whole paragraph) is aligned and not only the line the cursor is currently located.

Customization The HtmlArea offers you two properties to customize the paragraph handling globally and thus customize the behaviour of the component.

insertParagraphOnLinebreak The default value of this property is `true`. It controls whether a new paragraph or a normal line-break is inserted when hitting the `Enter` key. Since the default behaviour of all word processors is to insert a new paragraph it is recommended to leave this property value with its default.

Note: As every word processor the HtmlArea also supports inserting a normal line-break by using the key combination `Shift+Enter`

insertLinebreakOnCtrlEnter This property also has a default value of `true`. Since some users are familiar with the key combination `Control+Enter` to insert a normal line-break the HtmlArea component does support this. So in the default setup `Control+Enter` and `Shift+Enter` will end up with the same result.

Technical Background

Paragraph-Handling in Firefox

Browser control Currently the HtmlArea does only take control and manage the paragraphs on its own if

- SHIFT and CTRL keys are not pressed
- caret is not within a word
- focus node is not an element (current line is not empty)
- the focus is inside a list

HtmlArea control If the HtmlArea with its paragraph handling takes control, the following actions are taken.

Phase 1: Collecting styles

- computed styles of the focus node are collected
- these styles are grouped in the correct order (e.g. special handling for text-decoration because the text-decoration is linked to the elements color value)

Phase 2: Style string creation

- a `style` attribute based on the computed styles is generated for the paragraph element -> only `margin`, `padding` and `text-align` can be applied at paragraph-level. All other styles need to be applied at span elements (=child elements)
- a string with nested `span` / `font` element string is created. This element string is applied to the paragraph element. The nested structure is necessary because some styles need to be applied in the right order

Phase 3: Nodes creation The following string is applied with the "insertHtml" command

- an empty `span` element with an ID
- a `p` element with the paragraph style
- the nested `span` / `font` string to reflect the formatting which can't be applied at paragraph level

Phase 4: Cleanup

- Gecko inserts a `p` element on his own even if we intercept. This element gets removed afterwards by selecting this paragragph and inserting an empty DIV element at the selection
- the ID of the empty `span` is removed (Gecko will remove an empty `span` then automatically)
- if an empty paragraph is detected it will be removed to avoid rendering problems

Reasons for own paragraph handling

- support to keep formatting across multiple paragraphs or lists
- keep the caret always inside a `p` element
- keep control of the kind of line-breaking which is inserted
- normalize line-breaking
- act like MS Word

Issues

- DOM manipulations **can** break Undo/Redo since Gecko is expecting a DOM node which does not exist anymore
- edge cases can occur which are not targeted yet
- future browser implementation can change and mess up the current implementation
- MS Word behaviour can not be achieved in a browser, yet

List Handling

The component offers `ordered` and `unordered` lists to group content.

If the user inserts a new list

- `Enter` on a non-empty item: inserts a new list item
- `Enter` on an empty item: stops the editing of list
- `Shift+Enter`: inserts a new line within the current list item

These actions/key bindings are reflecting the default behaviour of word processors.

4.2.13 Widget Reference

- *Widget reference*

4.3 Layouts

4.3.1 Layouting

Introduction

A Layout manager defines the strategy of how to position the child widgets of a parent widget. They compute the position and size of each child by taking the size hints and layout properties of the children and the size hint of the parent into account.

Whenever the size of one widget changes, the layout engine will ask the layout manager of each affected widget to recompute its children's positions and sizes. Layout managers are only visible through the effects they have on the widgets they are responsible for.

It is possible to place and size all children directly to static positions using setUserBounds as well, but this is quite uncommon and only used in very special cases. It is almost always better to position children using a layout manager.

The layout manager can be configured on any widget, but most classes only have the protected methods to control the layout. In fact it doesn't make sense to control the layout manager of a `Spinner`, `ComboBox`, etc. from outside. So this scenario is quite common. Some widgets however publish the layout API. One of them is the above mentioned *Composite* widget. It exposes the layout system and the whole children API.

The nature of layout managers is that each one has specialized options for its children. For example, one layout allows specifying a left position of a child in the canvas while another one works with rows and cells instead. Given this fact, the best place to handle these options is the layout itself. Every `LayoutItem` has the methods setLayoutProperties and getLayoutProperties. Through this API the layout properties can be configured independendly from the layout.

The validation of properties is lazy (compared to the classic qooxdoo properties). At the moment where a child with layout properties is inserted into a parent widget with a layout, these properties are checked against the rules of the layout. This validation is not possible earlier, e.g. at the definition of the *wrong* property, as at this moment the child may not have a parent yet.

To make layout properties available in a convenient fashion each add() has an optional second parameter: A map with all layout properties to configure. A basic example:

```
var canvas = new qx.ui.container.Composite(new qx.ui.layout.Canvas);
canvas.add(new qx.ui.form.Button("Say Hello"), {
  left : 20,
  top: 20
});
```

This example places a button at the position 20x20 of the composite created. As you can see, the `Composite` widget has a convenient way – using the constructor – to define the layout it uses.

Panes

Some widgets extend the `Composite` widget above. Typical examples here are:

- TabView Page
- Popup

These have the same API like the composite. A slightly other type are so-called composite-like widgets. These widgets offer the same type of children management and layout management to the outside, but they redirect these properties to an inner pane.

Typical widgets in this category are:

- Window
- GroupBox

Sensible defaults

By default, widgets are intelligently auto-sized. This means that most of the time you can create a widget and it will look nice. If you need greater control, you can override the defaults. Every property defined initially is also reconfigurable during the runtime of an application. When using layout managers any computed sizes are automatically refreshed and the arrangement of children is updated.

Every automatically detected size can be overridden. Common settings of a widget (or spacers) are configured through the widget itself. This for example includes properties like width or height. All these sizes are pixel values. Percent and other complex values are only supported by a few layout managers so these are implemented as layout properties (explained in detail later).

Automatic size detection means, that limits are detected as well. Any widget in qooxdoo knows how much it can shrink and how much it can grow without interfering the functionality. The application developer can override these min/max sizes as well. This is no problem as long as the new value is tougher than the automatically detected values (e.g. lower limit of maximum width). When overriding the automatic sizes to reduce the limits layout problems may occur. It is highly suggested to keep an eye on this to omit such scenarios.

One thing to keep in mind is that the `width` cannot override the `minWidth` or the `maxWidth`. Limitation properties may be overridden by the property itself, but not by the normal size property. The `minWidth` can override the minimal automatically detected size, but the `width` cannot. This decision makes the layout system more stable as unintended overrides of the limitations are omitted in most cases.

Often `width` and `height` are described as preferred sizes as the given size may not have an influence on the actual rendered size of the widget. Even if the `width` is configured by the user, this does not mean that the widget always get the desired width.

Growing & Shrinking

Dynamic GUIs often must work equally well in cases where not enough (or too much) room is available to render the GUI in the way meant by the developer. This may include simple cases where the size of tabs is reduced in order to handle the display of all open tabs without scrolling. More advanced cases are text which wraps to multiple lines depending on the available width (and this way influences the position of following children).

In the first case we often see that an application reduces the size of the label and uses an ellipsis symbol to show that the label was too long. This feature is built-in into both commonly used widgets: Label and Atom. When the underlaying layout ask to reduce the width (or the developer using the `width` property) the widget tries to solve the requirement dynamically. This certainly works for the height as well.

```
var label = new qx.ui.basic.Label().set({
  value: "A long label text which has not enough room.",
  width: 60
});
```

The second case is handled by the height for width support. Longly name but basically a really strong feature which is required quite often. It means that the height may depend on the actual width available. This especially makes sense for multi-line text where the wrapping may be influenced by the available width. The Label widget includes support for this feature when using the rich output mode (HTML content).

```
var label = new qx.ui.basic.Label().set({
  value: "A long label text with auto-wrapping. This also may
    contain <b style='color:red'>rich HTML</b> markup.",
  rich : true,
  width: 120
});
```

Finally this means that every widget can grow and shrink depending on the limitations given for the respective axis. Two easy accessors which disable growing or shrinking respectively are allowGrowX and allowShrinkX. When the growing is disabled the configured or automatically detected maximum size is ignored and configured to the preferred size. When the shrinking is disabled the configured or automatically detected minimum size is ignored and configured to the preferred size. Two convenient methods to controlling these features without knowing of the exact dimensions.

Overflow Handling

This leads to the next question: how to handle scenarios where the content needs more room than provided by the parent but should not shrink. This is a common case for data widgets like Lists or Trees. Both extend the AbstractScrollArea to provide scrollbars to handle overflowing content.

The `ScrollArea` itself renders scrollbars in a custom way. It does not use the native scrollbars nor the native overflowing capabilities of the browser. Benefits of this decision are:

- Scroll bars can be themed.
- Optimal integration into layout system.
- Own implementation overrides browser quirks

The scrollbars are controlable in a way that is comparable to CSS. It is possible to have both scrollbars marked as `auto` to automatically detect the needs of the content. Or any other combination where a scrollbar may be statically hidden or visible. Each bar can be controlled separately. It is possible to enable one scrollbar statically and make the other one auto-displayed and vice-versa.

```
var big = new qx.ui.form.TextArea;
big.setWidth(600);
big.setHeight(600);

var area = new qx.ui.container.Scroll;
area.setWidth(200);
area.setHeight(200);
area.add(big);
```

The `ScrollArea` provides all typically needed methods like scrollToX to scroll to an absolute position or scrollByX to scroll by the given amount. The widget also supports the scrolling of any child into the viewport. This feature is provided through the method scrollItemIntoView. It just needs any child of the widget (at any depth).

```
var list = new qx.ui.form.List();
var item;
for (var i=0; i<20; i++)
{
  item = new qx.ui.form.ListItem("Item #" + i);
  list.add(item);

  if (i == 12) {
    list.select(item);
```

```
  }
}
```

One really interesting aspect of these scrolling features is, that they work all the time, even if the widget is not yet rendered. It is possible to scroll any `ScrollArea` before even rendered. It is even possible to scroll any child into view without the whole parent being visible. This is quite useful for selection handling (selected items should be visible). Selections of a list for example can be modified during the normal application runtime and are automatically applied and scrolled correctly after the first appearance on the screen.

Layout Properties

While there are a few core layout features which are normally respected by most layouts like the margin and alignment properties (have a look to the LayoutItem for these), there are layout specific properties which only makes sense in conjunction with the specified layout as well. These properties are called layout properties in qooxdoo.

These properties are normally defined with the addition to the parent widget. The children handling normally allows a second optional parameter `options`. The layout properties are given through a simple map e.g.

```
parent.add(child, {left:20, top: 100});
```

This is still good readble and directly defines the properties where the children is added to the parent (and the parent's layout). While this is the common use pattern of layout properties in qooxdoo applications, it is still possible to define layout properties afterwards using setLayoutProperties. The first parameter is like the second parameter in `add` and accepts a map of layout properties.

Units of Layout Properties

Pixel

Usually all position and size values are defined as pixel values. For example the `left` and `top` layout properties of the `Basic` layout are defined as pixel values.

Flex

The flex value indicates the flexibility of the item, which implies how an item's container distributes remaining empty space among its children. Flexible elements grow and shrink to fit their given space. Elements with larger flex values will be sized larger than elements with lower flex values, at the ratio determined by the two elements. The actual flex value is not relevant unless there are other flexible elements within the same container. Once the default sizes of elements in a box are calculated, the remaining space in the box is divided among the flexible elements, according to their flex ratios. Specifying a flex value of `0` has the same effect as leaving the flex attribute out entirely.

The easiest use case is to make exactly one child consuming the remaining space. This is often seen in modern application. For example the location field in common browsers are automatically configured to behave like this. To do this add a flex value of `1` to the child. In order to make more children behave like this, one could make them flexible the same way. The available space is automatically allocated between all of them. As `flex` allows integer values it is also possible to define weighted values. A flex value of `2` means double importance over `1`. The result is that from 100 pixel remaining space and two flexible children the one with `2` gets about 66 pixel and the other one 33 pixel.

Please note that in shrinking mode flex has an analogous effect. As a flex value of `2` means doubled importance compared to `1` the child with `2` is shrunken less than the child with `1`.

In contrast to qooxdoo 0.7 `flex` values are supplemental to the normal size values of a widget. First all children are positioned using their regular size hints. If after this step the combined size of the children is larger or smaller than the available size the `flex` value defines by how much each widget is stretched or shrunken.

The `flex` property is supported by both Box Layouts, the Dock Layout and the Grid (for columns and rows).

In some way the SplitPane supports flex as well, but it behaves a bit different there as it is regarded as an alternative to the preferred size.

Percent

With the above mentioned `flex` feature the use of percents is quite uncommon in most qooxdoo applications. Still, there are some cases where it might be interesting to define percent locations or dimensions.

The Canvas Layout for example allows a child's position to contain a percent value (e.g. the layout property `left` could be configured to `20%`). When there are 1000 pixel available the so-configured child is placed at a left coordinate of 200 pixel. The final coordinate is automatically updated when the outer dimensions are modified.

The LayoutItem's dimension properties only support integer values. To use percentage dimensions some qooxdoo layout managers allow to define width and height using layout properties. This dimensions are then *higher* prioritized than the width and height configured in the child using the *normal* properties. The limitations defined through `minWidth` etc. are still respected by the layout manager. Percentage dimensions are useful to allocate a specific part of the available space to a given widget without being dependent on the configuration of the other children.

It is possible to combine `flex` with percent dimensions. This is good because it allows to define *approximations* like `3` times `33%` instead of being forced to fill the `100%` completely. With flex enabled the layout manager automatically arranges the children to fill the remaining pixels.

The effects of percentage dimensions in box layouts are comparable to the result of flex in a SplitPane. The resulting size is computed from the available space less all statically configured gaps like spacings or margins. Layout managers with support for percentage dimensions are the already mentioned Box Layouts, but also the Canvas Layout as well as the Dock Layout.

Pre-configured Widgets

There are a few containers in qooxdoo which use a predefined immutable layout for rendering their children. Currently these containers are included:

- *Scroll*: Provides auto-matic scrollbars for larger content. Does not influence the size of the content which is rendered at the preferred size. Allows scrolling of the content. Supports advanced features like offset calculation and scroll into view.
- *Stack*: Scales every widget to the available space and put one over another. Allows selection of which child should be visible. Used internally by TabView etc.
- *SlideBar*: Comparable to the Scroll Container but only provides automatic forward and backward arrows. Supports only one axis per instance: horizontal or vertical. Buttons are automatically displayed as needed. Supports automatic shrinking of the children (other than the Scroll Container).
- *SplitPane*: Divides the available space into two areas and provides a possibility to resize the panes for the user. Automatically respects the limitations of each child.

Visibility Handling

Every widget can be hidden and shown at any time during the application runtime. In qooxdoo each widget's visibility might have three values: `visible`, `hidden` or `excluded`. While `hidden` and `excluded` both makes a widget invisible there is still a difference: `excluded` ignores the widget in during the layout process while `hidden` simply hides the widget and keeps the room for the widget during the layout process.

The `visibility` property is not commonly used in qooxdoo applications.There are a few nice accessor methods for each widget:

- To check the status of a widget: `isVisible()`, `isHidden()` and `isExcluded()`
- To modify the visibility: `show()`, `hide()` and `exclude()`

Please note that for performance reasons invisible widgets are not rendered or updated to the DOM which means that especially initially invisible parts could improve the startup of a qooxdoo application e.g. alternate Tab Pages, closed Window instances, Menus, etc.

To work with multiple layers like in a Tab View it is suggested to use a Stack Container instead of doing the visibility management on the own.

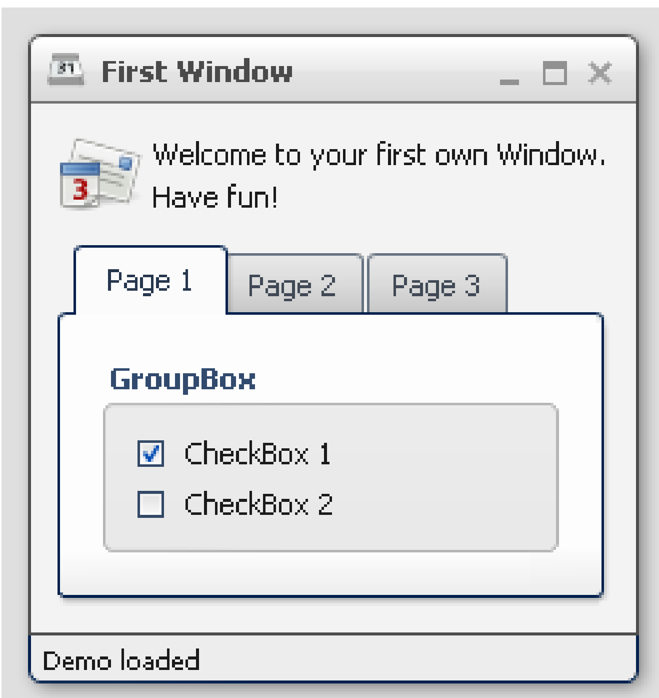
4.4 Themes

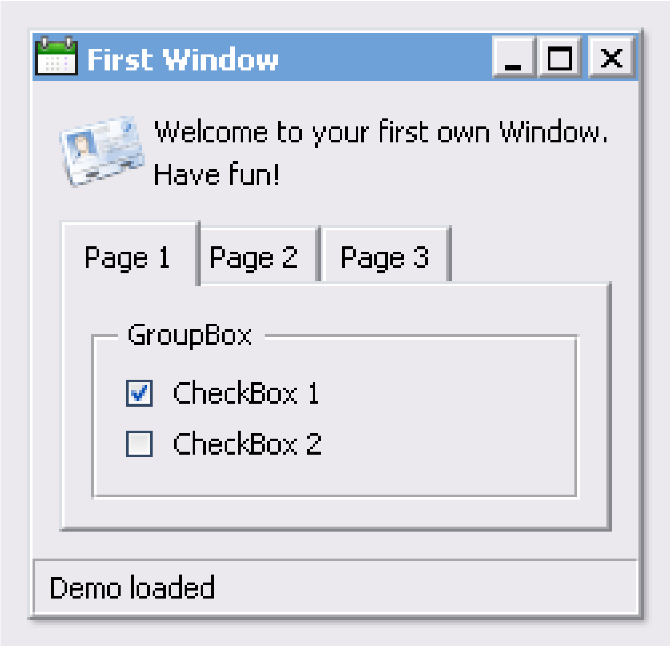
4.4.1 Theming

qooxdoo includes two themes:

- `Modern` - a graphically rich theme, showcasing many UI capabilities of qooxdoo 1.3.1
- `Classic` - a more lightweight, MS Windows oriented theme

Here some screenshots:





While those two themes run out-of-the-box, it is easy to create your own themes. Those custom themes can either be created by *extending existing ones* or they can be *created from scratch*.

A complete theme (a so-called *meta theme*) consists of several special themes, each designed to play a dedicated role and to setup the different parts of the whole theming. These special themes are described at the subsequent sections followed by a description of how to create own themes.

Meta Theme

A meta theme describes the whole theme itself by defining the specific parts. Each meta theme consists of five keys

- appearance
- color
- decoration
- font
- icon

each of them referencing to a specialized theme. So you can think of a meta theme as of collection whose parts can easily be changed.

Sample of a meta theme:

```
qx.Theme.define("qx.theme.Modern",
{
  meta :
  {
    color : qx.theme.modern.Color,
    decoration : qx.theme.modern.Decoration,
    font : qx.theme.modern.Font,
    appearance : qx.theme.modern.Appearance,
    icon : qx.theme.icon.Tango
  }
}
```

This section describes the different types of themes which are used for theming a whole application.

Color Theme

A color theme defines all colors used by the framework. Each color is defined by an unique name and a value which can be written as hex, rgb or named color. This defined name is usable throughout the whole framework and your application.

Note: The best way to organize your color names is to use **semantic ones** like `background`, `text-input` or `text-disabled`. This way it is easier to use one color for multiple widgets.

Part of a sample color theme:

```
/**
 * sample color theme
 */
qx.Theme.define("myApplication.theme.sample.Color",
{
  colors :
  {
    /*
    ---------------------------------------------------------------------------
      SAMPLE COLORS
    ---------------------------------------------------------------------------
    */

    // color as hex value
    "background-application" : "#DFDFDF",

    // color as rgb array
    "background-pane" : [ 128, 128, 128 ],

    // color as named color
    "background-light" : "gray",
  }
});
```

Following names are recognized as named colors: `black`, `white`, `silver`, `gray`, `maroon`, `red`, `purple`, `fuchsia`, `green`, `lime`, `olive`, `yellow`, `navy`, `blue`, `teal`, `aqua`, `orange`, `brown`.

The color values are set in the class qx.util.ColorUtil

Decoration Theme

Each widget can be equipped with an independent decoration which can be used to set a background-color or -image, define a border or add a shadow. In a decoration theme you can use several different decorators depending on the results you wish to achieve. Please take a look at the *decorator article* to get more information.

Note: It is recommend to define the decorations inside the theme instead of creating manually decorator instances inside your application code. This way the created decorators can be used by multiple widgets.

What a decoration theme can look like:

```
/* ************************************************

#asset(sample/decoration/myDecorationTheme/*)

************************************************* */
```

```
/**
 * sample decoration theme.
 */
qx.Theme.define("myApplication.theme.sample.Decoration",
{
  aliases : {
    decoration : "myApplication/decoration/sample"
  },

  decorations :
 {
    "uniform" :
    {
      decorator: qx.ui.decoration.Uniform,

      style :
      {
        width : 1,
        color : "border-main"
      }
    },

    "single" :
    {
      decorator: qx.ui.decoration.Single,

      style :
      {
        width : 1,

        color : "red",
        colorLeft : "black",
        colorRight : "white",

        style : "solid"
      }
    },

    "double" :
    {
      decorator : qx.ui.decoration.Double,

      style :
      {
        width : 1,
        innerWidth: 1,
        color : [ "border-dark-shadow", "border-light", "border-light", "border-dark-shadow" ],
        innerColor : [ "border-dark", "border-light-shadow", "border-light-shadow", "border-dark" ]
      }
    },

    "background" :
    {
      decorator : qx.ui.decoration.Background,

      style :
      {
        backgroundImage  : "decoration/background.png",
```

```
        backgroundRepeat : "scale"
      }
    },

    "beveled" :
    {
      decorator : qx.ui.decoration.Beveled,

      style : {
        backgroundImage : "decoration/beveled.png",
        backgroundRepeat : "scale",
        outerColor : "border-main",
        innerColor : "white",
        innerOpacity : 0.5
      }
    },

    "grid" :
    {
      decorator : qx.ui.decoration.Grid,

      style :
      {
        baseImage : "decoration/pane/grid.png"
      }
    }
});
```

Noted the `#asset` at the top and the `aliases` key inside the theme declaration? This is needed to for the images used within the theme. A description of how to work with resources is available *here*.

Note: The `aliases` key is especially important when defining an own decorator theme. This entry does add a new alias at the `AliasManager` class and verifies that your images for the decoration theme are found by the `ResourceManager` which is working with the resolve URLs of the `AliasManager` class.

Font Theme

This theme is all about the information of the fonts used throughout your application. As the number of types/variants of fonts used with application isn't that big the font theme is normally a compact one.

Note: It is always a good idea to limit the number of types or variants of fonts to create a homogenous look.

To demonstrate how compact and powerful a font theme can look like, take a look at the **complete** font theme of the Modern theme:

```
/**
 * The modern font theme.
 */
qx.Theme.define("qx.theme.modern.Font",
{
  fonts :
  {
    "default" :
    {
      size : qx.bom.client.System.WINVISTA ? 12 : 11,
      lineHeight : 1.4,
      family : qx.bom.client.Platform.MAC ? [ "Lucida Grande" ] :
        qx.bom.client.System.WINVISTA ? [ "Segoe UI", "Candara" ] :
```

```
        [ "Tahoma", "Liberation Sans", "Arial" ]
    },

    "bold" :
    {
      size : qx.bom.client.System.WINVISTA ? 12 : 11,
      lineHeight : 1.4,
      family : qx.bom.client.Platform.MAC ? [ "Lucida Grande" ] :
        qx.bom.client.System.WINVISTA ? [ "Segoe UI", "Candara" ] :
        [ "Tahoma", "Liberation Sans", "Arial" ],
      bold : true
    },

    "small" :
    {
      size : qx.bom.client.System.WINVISTA ? 11 : 10,
      lineHeight : 1.4,
      family : qx.bom.client.Platform.MAC ? [ "Lucida Grande" ] :
        qx.bom.client.System.WINVISTA ? [ "Segoe UI", "Candara" ] :
        [ "Tahoma", "Liberation Sans", "Arial" ]
    },

    "monospace" :
    {
      size: 11,
      lineHeight : 1.4,
      family : qx.bom.client.Platform.MAC ? [ "Lucida Grande" ] :
        qx.bom.client.System.WINVISTA ? [ "Consolas" ] :
        [ "Consolas", "DejaVu Sans Mono", "Courier New", "monospace" ]
    }
  }
});
```

Icon Theme

This theme is to define which icon set is used and normally consists only of 3 main keys (title, resource and icons).

The important one is the `resource` key which points the generator to the location of the icon set. The `icon` alias, which is used to reference icons in qooxdoo applications, is set to the value of this key. The `icons` key is to define additional icons which are not part of the icon theme. As qooxdoo uses the free available Tango and Oxygen icon sets it is not necessary to extend these.

Complete code for the `tango` icon theme:

```
/**
 * Tango icons
 */
qx.Theme.define("qx.theme.icon.Tango",
{
  aliases : {
    "icon" : "qx/icon/Tango"
  },
  icons : {}
});
```

Appearance Theme

The appearance theme is by far the biggest theme. Its task is to describe every themable widget and their child controls. Since the widgets are styled using decorators, colors, fonts and icons the appearance theme uses the definitions of all the other themes namely the decoration, color, font and icon theme. You can think of the appearance theme as the central meeting point where the other themes (decorator, color, font and icon) get together.

To discover the power of the appearance theme please take a look at the *corresponding article* which should let you get an idea of the whole picture.

Applying Themes

Typically, your application will have a certain, pre-defined theme known *at build-time*. The best way to associate such a default outlook with your application is to use the config.json variable `QXTHEME` inside the "let" section. Setting this variable to a fully-qualified meta theme class lets the build process handle the proper inclusion and linkage of the theme classes automatically. E.g.:

```
...
QXTHEME : my.theme.Cool,
...
```

It is also possible to set a certain appearance *at runtime*:

```
qx.theme.manager.Meta.getInstance().setTheme(my.theme.Cool);
```

For appearance, color, border, icon and widget themes, you can use similar classes in the qx.theme.manager package.

4.4.2 Appearance

Theme Structure

A theme normally consists of a set of entries. Each entry has a key which is basically some kind of selector which matches to a specific widgets. Missing selectors are presented as a warning when developing with debug code enabled.

```
qx.Theme.define("qx.theme.modern.Appearance",
{
  appearances :
  {
    selector : entry,
    [...]
  }
});
```

Selectors

In the most basic form each selector is identical to an appearance ID. This appearance ID is the value stored in the `appearance` property (API) of each widget.

The child control system ignores this appearance entry for widgets which function as a child control of another widget. In these cases the selector is the combination of the appearance ID of the parent widget plus the ID of the child control.

In a classic `Button` there is a child control `icon` for example. The appearance selector for the image element which represents the icon is `button/icon`. As you can see the divider between the appearance ID and the child control is a simple slash (`/`).

It is also possible that a widget, which is a child control itself, uses another child control. Generally the mechanism prepends the ID of each parent which is also a child control to the front of the selector. For example:

```
- pane
  - level1
    - level2
      - level3
```

the generated selector would be `pane/level1/level2/level3`. For `pane` which is not a child control of any other widget the appearance ID is used. For all others the child control ID is used. Again `pane` is not managed by any other widget so it is basically added by the developer of the application to another widget while `level1` to `level3` are managed by some type of combined widget and are added to each other without the work of the application developer.

A classic example for this is the `Spinner` widget. A `Spinner` is basically a Grid layout with a `TextField` and two `RepeatButtons`. The three internal widgets are available under the sub control IDs `textfield`, `upbutton` and `downbutton`. The selectors for these kind of child controls are then:

- `spinner/textfield`
- `spinner/upbutton`
- `spinner/downbutton`

Each of these selectors must be defined by the selected appearance. Otherwise a warning about missing selectors is displayed.

Aliases

A entry can be defined with two different values, a string or a map. The first option is named "alias", it is basically a string, redirecting to another selector. In the `Spinner` example from above we may just want to use aliases for the buttons. See the example:

```
qx.Theme.define("qx.theme.modern.Appearance",
{
  appearances :
  {
    [...],

    "spinner/upbutton" : "button",
    "spinner/downbutton" : "button",

    [...]
  }
});
```

So we have mastered one essential part for appearance themes. It is basically the easiest part, but seen quite often. Compared to CSS you always have a full control about the styling of such an child control. There is no type of implicit inheritance. This may also be seen negatively, but most developers tend to like it more.

Such an alias also redirects all child controls of the left hand selector to the right hand selector. This means that the icon inside the button is automatically redirected as well. Internally this mapping looks like this:

```
"spinner/upbutton" => "button"
"spinner/upbutton/icon" => "button/icon"
"spinner/upbutton/label" => "button/label"
```

This is super convenient for simple cases and additionally it is still possible to selectively override definitions for specific child controls.

```
qx.Theme.define("qx.theme.modern.Appearance",
{
  appearances :
  {
    [...],

    "myimage" : [...],

    "spinner/upbutton" : "button",
    "spinner/upbutton/icon" : "myimage",

    [...]
  }
});
```

Internally the above results into the following remapping:

```
"spinner/upbutton" => "button"
"spinner/upbutton/icon" => "myimage"
"spinner/upbutton/label" => "button/label"
```

Entries

The more complex full entry is a map with several sub entries where all are optional:

```
qx.Theme.define("qx.theme.modern.Appearance",
{
  appearances :
  {
    [...],

    "spinner/textfield" :
    {
      base : true/false,
      include : String,
      alias : String,

      style : function(states)
      {
        return {
          property : states.hovered ? value1 : value2,
          [...]
        };
      }
    },

    [...]
  }
});
```

Style Method

Let's start with the `style` sub entry. The value under this key should be a function which returns a set of properties to apply to the target widget. The first parameter of the function is named `states`. This is a map containing keys with boolean values which signalize which states are switched on. The data could be used to react on specific states like `hovered`, `focused`, `selected`, etc.

It is required that all properties applied in one state are applied in all other states. Something like this is seen as bad style and may result in wrong styling:

```
style : function(states)
{
  var result = {};

  if (states.hovered) {
    result.backgroundColor = "red";
  }
  // BAD: backgroundColor missing when widget isn't hovered!

  return result;
}
```

Instead, you should always define the else case:

```
style : function(states)
{
  var result = {};

  if (states.hovered) {
    result.backgroundColor = "red";
  } else {
    // GOOD: there should be a setting for all possible states
    result.backgroundColor = undefined;
  }

  return result;
}
```

Note: The `undefined` value means that no value should be applied. When qooxdoo runs through the returned map it calls the `reset` method for properties with a value of `undefined`. In most cases it would be also perfectly valid to use `null` instead of `undefined`, but keep in mind that `null` is stored using the setter (explicit null) and this way it overrides values given through the inheritance or through the init values. In short this means that `undefined` is the better choice in almost all cases.

One thing we have also seen in the example is that it is perfectly possible to create the return map using standard JavaScript and fill in keys during the runtime of the `style` method. This allows to use more complex statements to solve the requirements of today's themes were a lot of states or dependencies between states can have great impact on the result map.

Includes

Includes are used to reuse the result of another key and merge it with the local data. Includes may also used standalone without the `style` key but this is merly the same like an alias. An alias is the faster and better choice in this case.

The results of the include block are merged with lower priority than the local data so it just gets added to the map. To remove a key from the included map just define the key locally as well (using the `style` method) and set it to `undefined`.

Includes do nothing to child controls. They just include exactly the given selector into the current selector.

Child Control Aliases

Child control aliases are compared to the normal aliases mentioned above, just define aliases for the child controls. They do not redirect the local selector to the selector defined by the alias. An example to make this more clear:

```
qx.Theme.define("qx.theme.modern.Appearance",
{
  appearances :
  {
    [...],

    "spinner/upbutton" :
    {
      alias : "button",

      style : function(states) {
        return {
          padding : 2,
          icon : "decoration/arrows/up.gif"
        }
      }
    },

    [...]
  }
});
```

The result mapping would look like the following:

```
"spinner/upbutton" => "spinner/upbutton"
"spinner/upbutton/icon" => "button/image"
"spinner/upbutton/label" => "button/label"
```

As you can see the `spinner/upbutton` is kept in its original state. This allows one to just refine a specific outer part of a complex widget instead of the whole widget. It is also possible to include the orignal part of the `button` into the `spinner/upbutton` as well. This is useful to just override a few properties like seen in the following example:

```
qx.Theme.define("qx.theme.modern.Appearance",
{
  appearances :
  {
    [...],

    "spinner/upbutton" :
    {
      alias : "button",
      include : "button",

      style : function(states)
      {
        return {
          padding : 2,
          icon : "decoration/arrows/up.gif"
        }
      }
    },

    [...]
  }
```

```
});
```

When `alias` and `include` are identically pointing to the same selector the result is identical to the real alias

Base Calls

When extending themes the so-named `base` flag can be enabled to include the result of this selector of the derived theme into the local selector. This is quite comparable to the `this.base(arguments, ...)` call in member functions of typical qooxdoo classes. It does all the things the super class has done plus the local things. Please note that all local defintions have higher priority than the inheritance. See next paragraph for details.

Priorities

Priority is quite an important topic when dealing with so many sources to fill a selector with styles. Logically the definitions of the `style` function are the ones with the highest priority followed by the `include` block. The least priority has the `base` flag for enabling the *base calls* in inherited themes.

States

A state is used for every visual state a widget may have. Every state has flag character. It could only be enabled or disabled via the API `addState` or `removeState`.

Performance

qooxdoo has a lot of impressive caching ideas behind the whole appearance handling. As one could easily imagine all these features are quite expensive when they are made on every widget instance and more important, each time a state is modified.

Appearance Queue

First of all we have the appearance queue. Widgets which are visible and inserted into a visible parent are automatically processed by this queue when changes happen or on the initial display of the widget. Otherwise the change is delayed until the widget gets visible (again).

The queue also minimizes the effect of multiple state changes when they happen at once. All changes are combined into one lookup to the theme e.g. changing `hovered` and `focused` directly after each other would only result into one update instead of two. In a modern GUI typically each click influence a few widgets at once and in these widgets a few states at once so this optimization really pays of.

Selector Caching

Each widget comes with an appearance or was created as a child control of another widget. Because the detection of the selector is quite complex with iterations up to the parent chain, the resulting selector of each widget is cached. The system benefits from the idea that child controls are never moved outside the parent they belong to. So a child controls which is cached once keeps the selector for lifetime. The only thing which could invalidate the selectors of a widget and all of its child controls is the change of the property `appearance` in the parent of the child control.

Alias Caching

The support for aliases is resolved once per application load. So after a while all aliases are resolved to their final destination. This process is lazy and fills the redirection map with selector usage. This means that the relatively complex process of resolving all aliases is only done once.

The list of resolved aliases can be seen when printing out the map under `qx.theme.manager.Appearance.getInstance().__aliasMap` to the log console. It just contains the fully resolved alias (aliases may redirect to each other as well).

Result Caching

Further the result of each selector for a specific set of states is cached as well. This is maybe the most massive source of performance tweaks in the system. With the first usage, qooxdoo caches for example the result of `button` with the states `hovered` and `focused`. The result is used for any further request for such an appearance with the identical set of states. This caching is by the way the most evident reason why the appearance has no access to the individual widget. This would torpedate the caching in some way.

This last caching also reduces the overhead of `include` and `base` statements which are quite intensive tasks because of the map merge character with which they have been implemented.

4.4.3 Custom Themes

There are certain circumstances when the built-in themes are no more sufficient for your application and your needs. You need to create a custom theme because you have either self-written widgets you wish to style or you like to change the theming of your application overall.

Basically you have two choices to create a custom theme depending on your needs and the amount you want to change. The next two sections describe both briefly.

Extending Themes

If you want to stick with an existing theme and only like to add or modify some appearances, change colors or or fonts the best way to go is to extend a theme and to create an own meta theme which sets your extended theme.

For example you like to add some appearances (of your own widgets) to the Modern theme you can simply extend the appearance theme of the Modern theme.

```
qx.Theme.define("myApplication.theme.Appearance",
{
  extend : qx.theme.modern.Appearance,
  title : "my appearance theme",

  appearances :
  {
    "my-widget" :
    {
      alias : "atom",

      style : function(states)
      {
        return {
          width : 250,
          decorator : "main"
```

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

To enable your own appearance theme you also have to extend the Meta theme and set your appearance theme.

```
qx.Theme.define("myApplication.theme.Theme",
{
  title : "my meta theme",

  meta :
  {
    color : qx.theme.modern.Color,
    decoration : qx.theme.modern.Decoration,
    font : qx.theme.modern.Font,
    icon : qx.theme.icon.Tango,
    appearance : myApplication.theme.Appearance
  }
});
```

At last you have to tell the generator to actually use your meta theme. Therefore you have to edit your `config.json` file and add/edit the key `QXTHEME` in the `let` block.

```
"let" :
  {
    "APPLICATION"  : "myApplication",
    ...
    "QXTHEME"      : "myApplication.theme.Theme"
    ...
  },
```

After editing your `config.json` the very last step is to generate your application sources and you're done. Now you can adjust and extend your appearance theme to suit your needs.

Note: These steps are also applicable for the other themes.

Define Custom Themes

A custom theme is an own meta theme and the corresponding themes build from scratch. The main part of this work is mainly the appearance theme and the content of the other themes is mostly defined by the appearance theme, since this theme is the one who uses fonts, icons, decorators and colors.

Creating the meta theme is a no-brainer and when creating the several themes you only have to consider some rules:

- every theme has its own root key which also defines its type. `colors` for a color theme, `appearances` for an appearance theme and so on
- every widget has to be equipped with an appearance, otherwise you'll get a warning at application startup
- every used color, decorator or font has to be defined, otherwise you'll get an error at application startup. So be sure to define all used colors, fonts and decorators and to test your application always in the source version to get the error messages
- be sure to include every image you use in your appearance theme by defining corresponding `#asset` directives.

4.4.4 Decorators

Introduction

Decorations are used to style widgets. The idea is to have an independent layer around the widget content that can be freely styled. This way you can have separate decorators that define all kinds of decoration (colors, background image, corners, ...), and apply them to existing widgets, without interfering with the widget code itself.

Decorations are used for both, the `shadow` and the `decorator` property. They could be applied separately or together. There is no dependency between them.

Using Decorators

Generally all decorators used should be part of the selected decorator theme. The convention is that each decorator instance is stored under a semantic name. To use names which describe the appearance of the decorator is bad because it may make themes less compatible to each other.

It is also regarded as bad style to make use of so-named inline decorators which are created by hand as part of a function call. The reason for this is that generally decorators defined by the theme may be used in multiple places. This means that widgets and application code should not directly deal with decorator instances.

Custom Decorators

Custom decorators are created by extending the decorator theme and adding new ones or overwriting existing ones. Each decorator class comes with a set of properties for configuration of the instance. Following a short description of the available decorators:

- **Background**: Renders a background image or color
- **Uniform**: Like `Background`, but adds support for a uniform border which is identical for all edges.
- **Single**: Like `Background`, but adds support for separate borders for each edge.
- **Double**: Like `Single` but with the option to add two separate border to each edge.
- **Beveled**: Pseudo (lightweight) rounded border with support for inner glow. May contain a background image / gradient.
- **Grid**: Complex decorator based on nine images. Allows very customized styles (rounded borders, alpha transparency, gradients, ...). Optionally make use of image sprites to reduce image number.

Each entry of the theme is automatically made available using the `setDecorator`/`setShadow` functions of the widget class. The instances needed are automatically created when required initially. This mechanism keeps instance numbers down and basically ignores decorators which are defined but never used.

Writing Decorators

It is easily possible to write custom decorators. The interface is quite trivial to implement. There are only five methods which needs to be implemented:

- `getInsets`: Returns a map of insets (space the decorator needs) e.g. the border width
- `getMarkup`: Returns the initial markup needed to build the decorator. This is executed by each widget using the decorator. This method may not be used by some decorators and this way is defined as an empty method.
- `init`: Normally used to initialize the given element using `getMarkup`. Only executed once per element (read per widget).

- `resize`: Resizes the given element to the given dimensions. Directly works on the DOM to manipulate the content of the element.
- `tint`: Applies the given background color or resets it to the (optionally) locally defined background color. This method may not be used by some decorators and this way is defined as an empty method.

One thing to additionally respect is that `resize` and `tint` should be as fast as possible. They should be as minimal as possible as they are executed on every switch to the decorator (e.g. hover effects). All things which are possible to do once, in `getMarkup` or `init` methods, should be done there for performance reasons. Decorators are regarded as imutable. Once they are used somewhere there is no need to be able to change them anymore.

Each decorator configuration means exactly one decorator instance (created with the first usage). Even when dozens of widgets use the decorator only one instance is used. To cache the markup is a good way to improve the initial time to create new element instances. These configured elements are reused e.g. a hover effect which moves from “Button 1” to “Button 2” uses the same DOM element when reaching “Button 2” as it has used in “Button 1”. This way the number of DOM elements needed is reduced dramatically. Generally each decorator instance may be used to create dozens of these elements, but after some time enough elements may have been created to fulfill all further needs for the same styling.

4.4.5 Using themes of contributions in your application

Note: This tutorial assumes you are using the latest GUI skeleton template which contains pre-defined theme classes.

Contributions are a powerful and easy way to enhance your application with e.g. widgets that had not (yet) found the way into the qooxdoo core framework. Nevertheless it is a no-brainer to use them in your application.

But if a contribution is providing its own theme (in most cases its own appearance theme) you have to manage this manually.

Note: A bug report to make this step superfluous is already filed (see #1591), but in the meantime you can stick with this little tutorial to get things done.

For an easiser understanding this tutorial explains the necessary setup at the example of the TileView widget.

Adjust your configuration

The interesting part of the `config.json` looks like this:

```
...
"jobs" :
  {
    "libraries" :
    {
      "library" :
      [
        {
          "manifest" : "contrib://TileView/trunk/Manifest.json"
        },
       // as the tileView uses internally the FlowLayout you have
       // to add this to set it up correctly
       {
          "manifest" : "contrib://FlowLayout/trunk/Manifest.json"
        }
      ]
    }
  }
...
```

Include appearance theme

If you use the latest GUI skeleton template you will get an own appearance theme class (among all other theme classes) already setup for you. All you need to do is to include the appearance class provided by the `TileView` widget into your own appearance class.

Include the `TileView` appearance:

```
qx.Theme.define("yourApp.theme.Appearance",
{
  extend : qx.theme.modern.Appearance,

  // this include key does the magic
  include : tileview.theme.Appearance,

  // overwrite the appearances to customize the look of the modern theme
  // usually not needed
  appearances :
  {
  }
});
```

So all you need to add is this little `include` key with the corresponding appearance class to include it into your application.

Known issues

The following code which could reside in your `Application` class **won't** work:

```
qx.Theme.include(qx.theme.modern.Appearance, tileview.theme.Appearance);
```

The reason is that this include above will be resolved at **runtime** which does not work anymore. The first solution is resolved at **loading time**, so the include is already performed at startup. This issue is already filed under #1604.

CHAPTER

FIVE

LOW LEVEL FRAMEWORK

5.1 General

5.1.1 Overview

- Overview

5.1.2 Scenarios

- Scenarios

5.2 Tutorials

5.2.1 Setup a low-level library

A low-level library is interesting for all those who like to use the *low-level APIs* of qooxdoo. Such a library consists of a pre-build javascript file that contains only the low-level classes of qooxdoo. For instance, no GUI toolkit (widgets, layouts, theming) is included.

Create a low-level skeleton

To create your low-level application skeleton you can let do the tool-chain the heavy lifting and use the `create-application.py` script to generate the skeleton.

```
$QOOXDOO_PATH/tool/bin/create-application.py -n appName -t bom -o $OUTPUT-DIR
```

The `t` parameter is the important one to define the application as a `bom` type application. To show all available options of this mighty script just type

```
$QOOXDOO_PATH/tool/bin/create-application.py ?
```

Generate qooxdoo build

Looking at the output of your generated low-level library skeleton you first realize that no `source` folder exists. The simple reason for this is, that you can easily use the low-level APIs without creating your own application classes. Instead you add your logic directly into the given `index.html` or in whatever HTML file you like to.

Before you can descend to the low-levels you have to generate a javascript file containing the qooxdoo low-level classes.

```
./generate.py build
```

This pre-defined job is all you have to execute to start right away.

Note: The generated build script is a compilation of low-level classes, but it does **not** provide all classes of the *qx.bom* or *qx.dom* namespace. Please take a look at the provide *config.json* file to determine which file is included. The low-level wrapper of the XmlHttpRequest object (*qx.bom.Request*) is **not** provided by default.

Ready to code

As already mentioned implementing your logic is a no-brainer. Just grab the existing `index.html` file and start right away.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.2//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <script src="qx-bom.js" type="text/javascript" charset="utf-8"></script>
  <script type="text/javascript">
    // get informed about startup
    qx.event.Registration.addListener(window, "ready", onReady);

    function onReady(e)
    {
        <!-- your application code resides here -->
    }
  </script>
</head>
<body>
  <!-- more HTML -->
</body>
</html>
```

5.2.2 Low-Level APIs

This document describes the functionality of the low-level API classes in:

- qx.bom
- qx.dom
- qx.xml

qx.bom - Browser Object Model

The classes contained in the `qx.bom` namespace provide a cross-browser abstraction layer for object classes of the browser JavaScript runtime.

Note: This layer is heavily used by higher-level classes but can also be used stand-alone for low-level manipulations.

The BOM classes mainly consists of the following three parts:

- DOM element manipulation
- wrappers for native layers/objects

- powerful low-level helper classes

See the API reference of qx.bom for more details.

DOM element manipulation

The `qx.bom.element` package allows you to manipulate DOM elements in almost any way you can think of. Each class is offering several `static` methods that take a DOM element as their first argument. Since those BOM classes are static, no instances need to be created in order to manipulate a DOM element in the document.

The following manipulations are offered by the `qx.bom.element` package:

- Dimension and location
- Box-sizing - supports the modes `content-box` (W3C model) and `border-box` (Microsoft model)
- Scroll and overflow
- Style querying and modification
- CSS class name support - supports multiple class names for each element
- Scroll elements into view
- powerful low-level decoration support
- cross-browser support for opacity - optimized for animations
- Attribute/Property handling
- Background images and support for the clip property
- Cursor property

Wrapper for native layers/objects

These classes are offer an unique and powerful way to deal with native layers and objects. Wrappers exist for:

- the current document
- DOM elements to be connected to qooxdoo's event system
- native event management
- flash embedding
- CSS font styles
- several native controls like `iframe`, `form elements`, `label` and `image` elements

As every object or layer is abstracted by a corresponding qooxdoo class you can use these BOM classes to interact without worrying about the underlying browser used.

Additional classes

These additional classes help in developing low-level, cross-browser applications.

Features include:

- unified XMLHttp transport implementation
- powerful client detection classes

- low-level `Range` and `Selection` API
- helper class for browser history
- wrapper for working with CSS stylesheets
- string utility class
- helper class for the client's viewport
- helper class for VML

qx.dom - Cross-browser DOM manipulation

The Document Object Model (DOM) is a tree model that represents the document in a browser. The classes provided by this packages allow you to query, to manipulate (i.e. add, remove, change order or replace) and to check the nodes contained in the DOM.

Currently the `qx.dom` package consists of three classes:

- **Element**: manages children structures, inserts, removes and replaces nodes
- **Hierarchy**: for querying nodes
- **Node**: basic node creation and type detection

See the API reference of qx.dom for more details.

qx.xml - XML handling

This package is all about working with XML documents in a cross-browser way. Its three classes are:

- **Document**: creating an XML document
- **Element**: API to select, query and serialize XML elements
- **String**: escaping and unescaping of XML strings

See the API reference of qx.xml for more details.

5.2.3 Back-Button and Bookmark Support

Overview

Many Ajax applications break the browser back button and bookmarking support. Since the main page is never reloaded, the URL of the application never changes and no new entries are added to the browser history.

Fortunately it is possible to restore the expected behavior with a JavaScript history manager like the one included with qooxdoo (qx.bom.History).

Adding History support to an Application

To add history support to an application four basic steps are required:

- identify application states
- retrieve initial application state
- add event listener to history changes

- update history on application state changes

Identify Application States

The first step to add history support to an Ajax application is to identify the application states, which should be added to the history. This state must be encoded into a string, which will be set as the fragment identifier of the URL (the part after the '#' sign).

What exactly the application state is depends on the application. It can range from coarse grained states for basic application navigation to fine grained undo/redo steps. The API viewer uses e.g. the currently displayed class as its state.

Retrieve Initial Application State

At application startup the initial state should be read from the history manager. This enables bookmarks to specific states of the application, since the state is encoded into the URL. The URL `http://api.qooxdoo.org#qx.bom.History` would for example open the API viewer with the initial state of `qx.client.History`.

This is the code to read the initial state (getState API documentation):

```
var state = qx.bom.History.getInstance().getState();
```

Add Event Listener to History Changes

Each time the history changes by hitting the browser's back or forward button, the history manager dispatches a `request` event. The event object holds information about the new state. The application must add an event listener to this event and update the application state (request API documentation):

```
qx.bom.History.getInstance().addListener("request", function(e)
{
  var state = e.getData();

  // application specific state update
  this.setApplicationState(state);
}, this);
```

Update History on Application State Changes

Every time the application state changes, the history manager must be informed about the new state. A state change in the API viewer would for example occur if the user selects another class (addToHistory API documentation).

```
qx.bom.History.getInstance().addToHistory(state, title);
```

The first parameter is the state encoded as a string, which will be set as the URL fragment identifier. The second parameter is optional and may contain a string, which is set as the title of the browser window for this state.

5.3 Technical Topics

5.3.1 HTML Element Handling

This document describes the ideas and concepts behind the classes in the `qx.html` namespace (API). qooxdoo also comes with a basic low-level abstraction API for DOM manipulation. For details about this API please have a look at the *corresponding documentation*.

Idea

The classes in `qx.html` are wrapper for native DOM elements, which basically were created to solve one major issue:

Automatically keeping care of DOM manipulation and creation while dealing with large number of elements.

In details this means:

- **Automatic performance**: Programmatically constructing DOM hierarchies is hard to get fast because the order in which elements are nested can heavily influence the runtime performance. What `qx.html.Element` does is trying to keep the number of element instances to the minimum actually needed (DOM nodes are expensive, both performance and memory aside) and to insert the DOM nodes in an efficient manner. Further all changes to the DOM are cached and applied in batch mode, which improves the performance even more.
- **Normalized API**: Working with HTML DOM elements usually involves many browser switches. Especially when it comes to reading and setting of attributes or styles. For each style one has to remember whether a normalization method should be called or if the value can be set directly. `qx.html.Element` does this kind of normalization transparently. The browser normalization is based on the *existing low-level APIs*.
- **Convenience methods**: These elements have additional convenience API, which is not available on pure DOM elements. They have e.g. the functionality to manage children with methods like `addBefore()` or `moveAfter()`.

Typical Use Cases

- Building a widget system on top
- Massively building DOM elements from data structures

It may be used for smaller things as well, but brings in quite some overhead. The size of the API, additional to a basic low-level package of qooxdoo is about 20 KB (5 KB gzipped). Also it consumes a bit more memory when all underlying DOM elements are created. Keep in mind that the instances are around all the time. Even when all jobs for a instance are done at the moment.

Features

- Automatic DOM insertion and element management
- Full cross-browser support through usage of low-level APIs e.g. `setStyle()`, `getAttribute()`, ...
- Advanced children handling with a lot of convenience methods e.g. `addAfter()`, ...
- Reuse existing markup as a base of any element via `useMarkup()`
- Reuse an existing DOM node via `useElement()`
- Powerful visibility handling to `include()` or `exclude()` specific sub trees

- Support for scrolling and scroll into view (`scrollTo()`, `scrollIntoView()`, ...)
- Integration of text selection APIs (`setSelection()`, `getSelection()`, ...)
- Automatic interaction with event managers (`addListener()`, `removeListener()`, ...)
- Connection to focus/activation handler

Specific HTML Elements

Roots

A root is one essential element type when dealing with the API. Every user of `qx.html.Element` needs at least one instance of `qx.html.Root` to insert children to it. The root is always marked as being visible and is typically the body DOM element or any other directly inserted element. This element can be assigned to be used by the root using the method `useElement()`.

Labels

Used for all types of text content. Supports text or HTML content togglable using the `setRich()` method. When using the text mode ellipsis is supports in all browsers to show an indication when the text is larger than the available space. Highly depends on the API of qx.bom.Label.

Images

An element pre-configured as a `IMG` tag. Supports scaled and unscaled images. Supports image clipping (without scaling) to more efficiently deal with a lot of images. Depends on the API brought in by qx.bom.element.Decoration. Input —–=

This element is used for all types of input fields. The type can be given using a constructor parameter. It allows configuration of the `value` and the text wrapping (requires type `textarea`). Depends on the API brought in by qx.bom.Input.

Iframe

This element is used to create iframes to embed content from other sources to the DOM. It wraps the features of qx.bom.Iframe. Supports to configure the source of the iframe as well as its name. Comes with accessors to the document or window object of the iframe.

Canvas

Renders a HTML5 Canvas to the DOM. Has methods to access the render context as well to configure the dimensions of the Canvas.

The Queue

Internally most actions applied to the instances of `qx.html.Element` are applied lazily to the DOM. All style or attribute changes are queued for example to set them at once. This is especially useful to allow to bump out changes at once to the browser even when these happens in multi places and more important on more than one element.

Even things like focus handling or scrolling may be queued. It depends on if the element is currently visibible etc. whether these are queued. `focus` makes often more sense when it is directly executed as the following code may make assumptions that the changes are applied already. Generally qooxdoo allows it to apply most changes without the queue as well using a `direct` flag which is part of most setters offered by `qx.html.Element`.

5.3.2 Image Handling

This document tries to give some insights into the low-level features for image handling. This includes the functionality of these classes:

- `qx.bom.element.Background` (API)
- `qx.bom.element.Decoration` (API)

Generally there are two common ways to show images in a browser: normal image elements and background images. The `Decoration` class supports both of them and automatically selects the type to use for a specific requirement. The `Background` class is a simple wrapper around the support for background images. It is mainly some cross-browser magic to fix a few quirks of some of the supported engines.

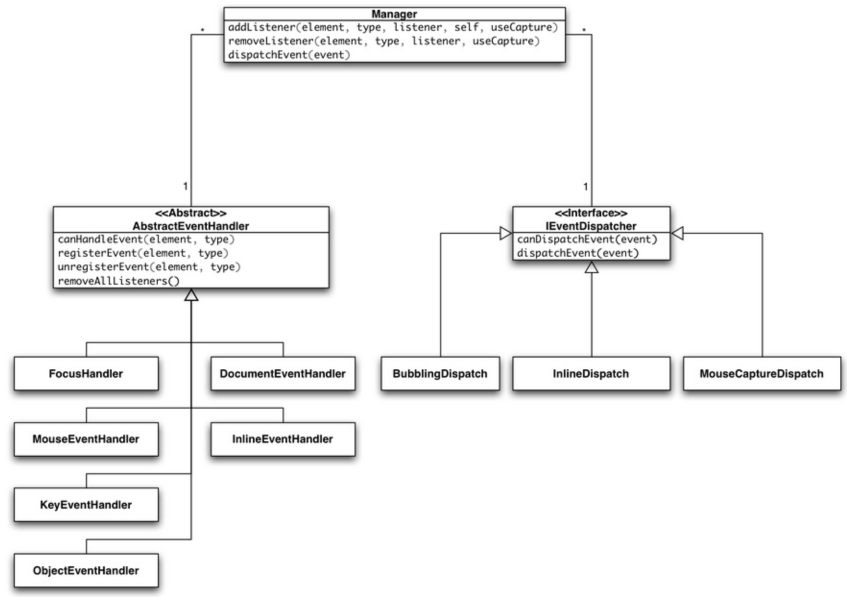
5.3.3 The Event Layer

The class qx.event.Manager provides a per-document wrapper for cross-browser DOM event handling. The implementation of the event layer is inside the qx.event namespace.

The following features work in all *supported browsers*:

- Canceling events: `stopPropagation()`
- Skipping the browser's default behavior: `preventDefault()`
- Unified event objects matching the W3C DOM 2 event interface
- Cross-browser event *bubbling* and *capturing* phase, even in Internet Explorer
- Mouse event capturing
- Port of the unified qooxdoo 0.7 key event handler to the 1.2 low-level layer. For a full list of available key identifiers see the getKeyIdentifier() method documentation of the `qx.event.type.KeySequence` class.
- Unified mouse events
  - Normalized double click event sequence `mousedown` -> `mouseup` -> `click` -> `mousedown` -> `mouseup` -> `click` -> `doubleclick` in Internet Explorer
  - Normalized right click sequence `mousedown` -> `mouseup` -> `contextmenu` in Safari 3 and Opera.
  - Always fire `click` events if the `mouseup` happens on a different target than the corresponding `mousedown` event. Natively only Internet Explorer behaves like that.

UML Class Diagram



5.3.4 The Focus Layer

History

This document is meant to talk about some internals of the focus system in qooxdoo since 1.2. This is a technology documentation targeted to interested developers. There is no need to understand these details as a user of the framework.

In previous versions of the focus handling we forced the application to our own implementation instead of working together with the browser. This was quite straightforward because the topic itself is quite complex and the differences between the browsers are huge. So just ignoring all these differences and implementing an own layer is highly attractive.

However this came with quite some costs. For example it's quite hard to catch all the edge cases when a input field loses the focus nor is it possible to recover the focus correctly when the browser does something after switching the window (send back/bring to front etc.). To listen on the browser might improve some types of out-of-sync problems in the previous versions. We caught most things correctly though, but it is quite hard to get 100% accuracy.

Focus Support

With 1.2 the focus system was reimplemented using the new low level event stack. Compared to the old focus system this basically means that the whole focus support is implemented low-level without any dependencies on the widget system. It directly uses the new event infrastructure and integrates fine with the other event handlers.

The new system tries to connect with all available native events which could help with detecting were the browser's focus is moving to. The implementation makes use of native events like `activate` or `focusin` where available. It uses a lot of browser behavior which is not explicitly documented or valid when reading the specifications, just to solve the issue of detecting where the focus currently is or is moved to.

It supports the events `focusin`, `focus`, `focusout` and `blur` on DOM nodes. It also supports `focus` and `blur` events on the `window`. There is support for `activate` and `deactivate` events on DOM nodes to track keyboard activation. It has the properties `focus` and `active` to ask for the currently focused or activated DOM node.

Activation Support

The activation, as part of the focus system, is also done by this manager. The keyboard handler for example asks the focus system which DOM element is the active one to start the bubble sequences for all keyboard events on this element. As the keyboard layer sits on top of the DOM and implements the event phases on its own there is no need to inform the browser about the active DOM node as it is simply not relevant when using this layer. It is also quite important as in every browser tested the methods to activate a DOM node (if available at all) might also influence the focus which creates some problems.

Window Focus/Blur

The handler also manages the focus state of the top-level window. It fires the `blur` and `focus` events on the `window` object one can listen to. Natively, these events are fired all over just by clicking somewhere in the document. The issue is to detect the *real* `focus`/`blur` events. This is implemented through some type of internal state representation.

Text Selection

Focus handling in qooxdoo also solves a lot of related issues. For example the whole support for unelectable text is done with the focus handler as well. Normally all text content on a page is selectable (with some exceptions like native form buttons etc.). In a typical GUI or during drag&drop sessions it is highly needed to stop the user from being able to select any text.

The only thing needed for the focus handler here is to add an attribute `qxSelectable` with the value `off` to the node which should not be selectable. I don't know about a way which is easier to solve this need.

Behind the scenes qooxdoo dynamically applies styles like `user-select` or attributes like `unselectable`. There are a lot of bugs in the browser when keeping these attributes or styles statically applied to the nodes so they are applied as needed dynamically which works surprisingly well. In Internet Explorer the handler stops the event `selectstart` for the affected elements.

Prevent Defaults

One thing we needed especially for the widget system, which is built on top, was support for preventing a widget or in this case a DOM node from being able to get the focus. This sounds simpler at first than it is. The major issue is to also keep the focus where it is while clicking somewhere else.

This is especially interesting when working with a text selection. Unfortunately in a browser the selection could only be where the focus is. This is a major issue when trying to apply any change to the currently selected text like needed for most kinds of editors (like a rich text editor used by a mail application for example). The type of fix we apply in qooxdoo is not to allow the browser to focus a specific DOM node e.g. the "Bold" button of the text editor. This makes it easy to add listeners to the button which work with the still existing selection of the editor field. The feature could be applied easily to a DOM node like such a button just through an attribute `qxKeepFocus` with the value `on`. It affects all children of the element as well, as long as these do not define anything else.

A similar goal is to keep the activation where it is when the user clicks at a specific section of the document. This is mainly used to keep the keyboard processing where it is e.g. when clicking the opened list of a `SelectBox` widget. This feature could be used for other scenarios like this as well. Like in the previous block it can be enabled simply by setting the attribute `qxKeepActive` to `on` for the relevant DOM node. Internally, to stop the activation also means to stop the focus. It was not solvable in another way because the browser otherwise sends activation events to the focused DOM node which is contra productive in this case.

Another unwanted side effect of some browsers is the possibility to drag around specific types of content. There is some type of native drag&drop support in most of today's browsers, but this is quite useless with the current quality of implementation. Still, the major issue remains: It is possible to drag around images for example which is often not wanted in a GUI toolkit. These native *features* compromise the behavior implemented by the application developer on top of them. To stop this, qooxdoo applies styles like `user-drag` on browsers that support it, or prevents the native `draggesture` event where available.

Other then this, most of these prevention is implemented internally through a `preventDefault` call on the global `mousedown` event when a specific target is detected. This has some side effects though. When preventing such a core event it means that most browsers also stop any type of selection happening through the mouse. This also stops them from focusing the DOM node natively. The qooxdoo code uses some explicit `focus` calls on the DOM nodes to fix this.

Please note that some settings may have side effects on other things. For example, to make a text region selectable but not activate able is not possible with the current implementation. This has not really a relevance in real-world applications, but may be still interesting to know about.

Finally

Finally, the whole implementation differs nearly completely for the supported browsers. Hopefully you get an impression of the complexity of the topic. May the browser with you.

5.3.5 qooxdoo Animation

qooxdoo Animation is a low level animation layer which comes with several effects to animate DOM elements. An effect changes one or more attributes of a DOM element from a start to an end value in the given time either linear or using a transition function. Effects can be stacked in a queue and orderd by assigning a startup delay.

- API documentation
- Demos
- Issues

Usage

To create an effect instance the desired effect and pass the DOM element, which should be used for the animation, as parameter. The effect can be configured by changing the properties like `from`, `to`, `duration` and more. Once the effect is set up, it can be started by calling the `start()` method of the effect.

```
var element = document.getElementById("dlAmount");
var attention = new qx.fx.effect.core.Highlight(element);

function update(amount)
{
  element.innerHTML = parseInt(amount);
  attention.start();
}
```

Queueing effecs

Every effect has a `delay` property which can be set to the amount of seconds the effect should wait before it should be executed after calling the `start()` method on it. You can use this property to arrange effects in the order you want them to be executed.

```
var el = button1.getContainerElement().getDomElement();

var psEffect = new qx.fx.effect.combination.Pulsate(el);

// The pulsate effect will take two seconds to execute
psEffect.seDuration(2);

var  mvEffect = new qx.fx.effect.core.Move(el);
mvEffect.set({
  x : 100,
  y : 200,
  delay : 2 // Wait two seconds to execute
});

// Start both effects at the same time
psEffect.start();
mvEffect.start();
```

Writing own effects

To create own effects, create a new class and extend from `qx.fx.Base` and overwrite the `update()` methode. You can access the DOM element of the effect by calling `this._getElement()`.

```
qx.Class.define("fxdemo.flickerBackground",
{
  extend : qx.fx.Base,

  members :
  {
    update : function(value)
    {
      var element = this._getElement();

      // Value is a floating-point number between the start and end property.
      value +=""; // Convert it to a string.
      value = parseInt(value[value.length-1], 10); // Read the last digit and parse it to integer
      element.style.backgroundColor = "'" + (value % 2 == 0) ? "red" : "blue" + "'";
    }
  }
});
```

List of effects

The `qx.fx.effect` package contains 14 effects:

- **ColorFlow** Changes the background color of an element to a given initial. After that the effects waits a given amount of time before it modifies to background color back to the initial value.
- **Drop** Moves the element to the given direction while fading it out.
- **Fade** Fades in the specified element: it changes to opacity from a given value to another. If target value is 0, it will hide the element, if value is 1, it will show it using the "display" property.
- **Fold** Shrinks the element in width and height until it gets invisible.
- **Grow** Resizes the element from initial dimensions to final dimensions.
- **Highlight** Cycles the background color of the element from initial to final color.

- **Move** Moves to element to the given coordinates.
- **Puff** Resizes the element from zero to the original size of the elment and fades it in at the same time.
- **Pulsate** Fades the element in and out several times.
- **Scale** This effect scales the specified element (and its content, optionally) by given percentages.
- **Scroll** Scrolls to specified coordinates on given element.
- **Shake** Moves the element forwards and backwards several times.
- **Shrink** Resizes the element from initial to given dimensions.
- **Switch** Flickers the element one time and then folds it in.

CHAPTER

SIX

COMMUNICATION

There are two forms of client-server communication supported:

6.1 Low-level AJAX Calls

6.1.1 AJAX

This system is (as everything else in qooxdoo) completely event based. It currently supports communication by **XMLHttp**, **Iframes** or **Script**. The system wraps most of the differences between the implementations and unifies them for the user/developer.

For all your communication needs you need to create a new instance of Request:

```
var req = new qx.io.remote.Request(url, "GET", "text/plain");
```

Constructor arguments of Request:

1. URL: Any valid http/https/file URL
2. Method: You can choose between POST and GET.
3. Response mimetype: What mimetype do you await as response

Mimetypes supported

- application/xml
- text/plain
- text/html
- text/javascript
- application/json

Note: `text/javascript` and `application/json` will be directly evaluated. As content you will get the return value.

If you use the iframe transport implementation the functionality of the type is more dependent on the server side response than for the XMLHttp case. For example the text/html mimetypes really need the response in HTML and can't convert it. This also depends greatly on the mimetype sent out by the server.

Request data

- `setRequestHeader(key, value)`: Setup a request header to send.
- `getRequestHeader(key)`: Returns the configured value of the request header.
- `setParameter(key, value)`: Add a parameter to send with your request.
- `getParameter(key)`: Returns the value of the given parameter.
- `setData(value)`: Sets the data which should be sent with the request (only useful for POST)
- `getData()`: Returns the data currently set for the request

Note: Parameters are always sent as part of the URL, even if you select POST. If you select POST, use the setData method to set the data for the request body.

Request configuration (properties)

- `asynchronous`: Should the request be asynchronous? This is `true` by default. Otherwise it will stop the script execution until the response was received.
- `data`: Data to send with the request. Only used for POST requests. This is the actual post data. Generally this is a string of url-encoded key-value pairs.
- `username`: The user name to authorize for the server. Configure this to enable authentication.
- `password`: The password to authenticate for the server.
- `timeout`: Configure the timeout in milliseconds of each request. After this timeout the request will be automatically canceled.
- `prohibitCaching`: Add a random numeric key-value pair to the url to securely prohibit caching in IE. Enabled by default.
- `crossDomain`: Enable/disable cross-domain transfers. This is `false` by default. If you need to acquire data from a server of a different domain you would need to setup this as `true`. (**Caution:** this would switch to "script" transport, which is a security risk as you evaluate code from an external source. Please understand the security issues involved.)
- `fileUpload`: Indicate that the request will be used for a file upload. The request will be used for a file upload. This switches the concrete implementation that is used for sending the request from `qx.io.remote.transport.XmlHttp` to `qx.io.remote.IFrameTransport`, because only the latter can handle file uploads.

Available events

- `sending`: Request was configured and is sending data to the server.
- `receiving`: The client receives the response of the server.
- `completed`: The request was executed successfully.
- `failed`: The request failed through some reason.
- `timeout`: The request has got a timeout event.
- `aborted`: The request was aborted.

The last four events give you a `qx.event.type.Data` as the first parameter of the event handler. As always for `qx.event.type.Data` you can access the stored data using `getData()`. The return value of this function is an instance of `qx.io.remote.Response`.

Response object

The response object `qx.io.remote.Response` stores all the returning data of a `qx.io.remote.Request`. This object comes with the following methods:

- `getContent`: Returns the content data of the response. This should be the type of content you acquired using the request.
- `getResponseHeader`: Returns the content of the given header entry.
- `getResponseHeaders`: Return all available response headers. This is a hash-map using typical key-values pairs.
- `getStatusCode`: Returns the HTTP status code.

Note: Response headers and status code information are not supported for iframe based communication!

Simple example

```
// get text from the server
req = new qx.io.remote.Request(val.getLabel(), "GET", "text/plain");
// request a javascript file from the server
// req = new qx.io.remote.Request(val.getLabel(), "GET", "text/javascript");

// Switching to POST
// req.setMethod("POST");
// req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

// Adding parameters - will be added to the URL
// req.setParameter("test1", "value1");
// req.setParameter("test2", "value2");

// Adding data to the request body
// req.setData("foobar");

// Force to testing iframe implementation
// req.setCrossDomain(true);

req.addListener("completed", function(e) {
  alert(e.getContent());
  // use the following for qooxdoo versions <= 0.6.7:
  // alert(e.getData().getContent());
});

// Sending
req.send();
```

Please post questions to our mailinglist.

6.2 Higher-level Remote Procedure Calls (RPC)

6.2.1 RPC (Remote Procedure Call)

qooxdoo includes an advanced RPC mechanism for direct calls to server-side methods. It allows you to write true client/server applications without having to worry about the communication details.

The qooxdoo RPC is based on JSON-RPC as the serialization and method call protocol, and qooxdoo provides server backends for Java, PHP, and Perl projects. A Python backend library is also provided by a third party. All parameters and return values are automatically converted between JavaScript and the server-side language.

Setup

To make use of the RPC, you need to set up a server backend first.

Configuration of each server backend needs slightly different treatment. Please see the page relevant to you:

- *Java*
- *PHP*
- *Perl*
- *Python*

Your favorite language is missing? Feel free to write your own qooxdoo RPC server, consult the *RPC Server Writer Guide* for details.

Making remote calls

Basic call syntax

To make remote calls, you need to create an instance of the Rpc class:

```
var rpc = new qx.io.remote.Rpc(
    "http://localhost:8080/qooxdoo/.qxrpc",
    "qooxdoo.test"
);
```

The first parameter is the URL of the backend (in this example a Java backend on localhost). The second is the name of the service you'd like to call. In Java, this is the fully qualified name of a class. (The Java backend includes the `qooxdoo.test` service used in the example. The class name is lowercase to keep it in sync with the PHP examples - in Java-only projects, you would of course use standard Java naming conventions.)

When you have the Rpc instance, you can make synchronous and asynchronous calls:

```
// synchronous call
try {
    var result = rpc.callSync("echo", "Test");
    alert("Result of sync call: " + result);
} catch (exc) {
    alert("Exception during sync call: " + exc);
}

// asynchronous call
var handler = function(result, exc) {
    if (exc == null) {
        alert("Result of async call: " + result);
    } else {
        alert("Exception during async call: " + exc);
    }
};
rpc.callAsync(handler, "echo", "Test");
```

For synchronous calls, the first parameter is the method name. After that, one or more parameters for this method may follow (in this case, a single string). Please note that synchronous calls typically block the browser UI until the result arrives, so they should only be used sparingly (if at all)!

Asynchronous calls work similarly. The only difference is an additional first parameter that specifies a handler function. This function is called when the result of the method call is available or when an exception occurred.

You can also use qooxdoo event listeners for asynchronous calls - just use `callAsyncListeners` instead of `callAsync`. More details can be found in the API documentation.

One difference between the qooxdoo RPC and other RPC implementations are client stubs. These are small wrapper classes that provide the same methods as the corresponding server classes, so they can be called like ordinary JavaScript methods. In qooxdoo, there are no such stubs by default, so you have to provide the method name as a string. The advantage is that there's no additional build step for generating stubs, and it's also not necessary to "register" your server classes at runtime (which would be a prerequisite for dynamic stub generation). If you really want or need client stubs, you currently have to write the stubs (or a generator for them) yourself. Future qooxdoo versions may include such a generator.

Parameter and result conversion

All method parameters and result values are automatically converted to and from the backend language. Using the Java backend, you can even have overloaded methods, and the correct one will be picked based on the provided parameters.

The following table lists the data types supported by the Java backend and the corresponding JavaScript types:

Java type	JavaScript type
int, long, double, Integer, Long, Double	number
boolean, Boolean	boolean
String	String
java.util.Date	Date
Array (of any of the supported types)	Array
java.util.Map	Object
JavaBean	Object

The first few cases are quite simple, but the last two need some more explanation. If a Java method expects a `java.util.Map`, you can send any JavaScript object to it. All properties of the object are converted to Java and become members of the Java Map. When a Map is used as a return value, it's converted to a JavaScript object in a similar way: A new object is created, and then all key/value pairs in the map are converted themselves and then added as properties to this object. (Please note that "properties" is used here in the native JavaScript sense, not in the sense of *qooxdoo properties*.)

JavaBeans are converted in a similar way. The properties of the JavaBean become JavaScript properties and vice versa. If a JavaScript object contains properties for which no corresponding setters exist in the JavaBean, they are ignored.

For performance reasons, recursive conversion of JavaBeans and Maps is performed without checking for cycles! If there's a reference cycle somewhere, you end up with a StackOverflowException. The same is true when you try to send a JavaScript object to the server: If it (indirectly) references itself, you get a recursion error in the browser.

Besides the fully-automatic conversions, there's also a class hinting mechanism. You can use it in case you need to send a specific sub-class to the server (see below for details). However, it can't be used to instantiate classes without a default constructor yet. Future qooxdoo versions may provide more extensive class hinting support.

Aborting a call

You can abort an asynchronous call while it's still being performed:

```
// Rpc instantiation and handler function left out for brevity

var callref = rpc.callAsync(handler, "echo", "Test");

// ...

rpc.abort(callref);
  // the handler will be called with an abort exception
```

Error handling

When you make a synchronous call, you can catch an exception to handle errors. In its `rpcdetails` property, the exception contains an object that describes the error in more detail. The same details are also available in the second parameter in an asynchronous handler function, as well as in the events fired by `callAsyncListeners`.

The following example shows how errors can be handled:

```
// creation of the Rpc instance left out for brevity

var showDetails = function(details) {
    alert(
        "origin: " + details.origin +
        "; code: " + details.code +
        "; message: " + details.message
    );
};

// error handling for sync calls
try {
    var result = rpc.callSync("echo", "Test");
} catch (exc) {
    showDetails(exc.rpcdetails);
}

// error handling for async calls
var handler = function(result, exc) {
    if (exc != null) {
        showDetails(exc);
    }
};
rpc.callAsync(handler, "echo", "Test");
```

The following `origin`‘s are defined:

Constant	Meaning
qx.io.remote.Rpc.origin.server	The error occurred on the server (e.g. when a non-existing method is called).
qx.io.remote.Rpc.origin.application	The error occurred inside the server application (i.e. during a method call in non-qooxdoo code).
qx.io.remote.Rpc.origin.transport	The error occurred in the communication layer (e.g. when the Rpc instance was constructed with an URL where no backend is deployed, resulting in an HTTP 404 error).
qx.io.remote.Rpc.origin.local	The error occurred locally (when the call timed out or when it was aborted).

The `code` depends on the origin. For the server and application origins, the possible codes are defined by the backend implementation. For transport errors, it’s the HTTP status code. For local errors, the following codes are defined:

Constant	Meaning
qx.io.remote.Rpc.localError.timeout	A timeout occurred.
qx.io.remote.Rpc.localError.abort	The call was aborted.

Cross-domain calls

Using the qooxdoo RPC implementation, you can also make calls across domain boundaries. On the client side, all you have to do is specify the correct destination URL in the Rpc constructor and set the crossDomain property to `true`:

```
var rpc = new qx.io.remote.Rpc("http://targetdomain.com/appname/.qxrpc");
rpc.setCrossDomain(true);
```

On the server side, you need to configure the backend to accept cross-domain calls (see the documentation comments in the various backend implementations).

Writing your own services

Java

Writing your own remotely callable methods is very easy. Just create a class like this:

```
package my.package;

import net.sf.qooxdoo.rpc.RemoteService;
import net.sf.qooxdoo.rpc.RemoteServiceException;

public class MyService implements RemoteService {

    public int add(int a, int b) throws RemoteServiceException {
        return a + b;
    }

}
```

All you need to do is include this class in your webapp (together with the qooxdoo backend classes), and it will be available for calls from JavaScript! You don't need to write or modify any configuration files, and you don't need to register this class anywhere. The only requirements are:

1. The class has to implement the `RemoteService` interface. This is a so-called tagging interface, i.e. it has no methods.
2. All methods that should be remotely available must be declared to throw a `RemoteServiceException`.

Both requirements are there to protect arbitrary Java code from being called.

Accessing the session There is one instance of a service class per session. To get access to the current session, you can provide an *injection* method called `setQooxdooEnvironment`:

```
package my.package;

import javax.servlet.http.HttpSession;

import net.sf.qooxdoo.rpc.Environment;
import net.sf.qooxdoo.rpc.RemoteService;
```

```
import net.sf.qooxdoo.rpc.RemoteServiceException;

public class MyService implements RemoteService {

    private Environment _env;

    public void setQooxdooEnvironment(Environment env) {
        _env = env;
    }

    public void someRemoteMethod() throws RemoteServiceException {
        HttpSession session = _env.getRequest().getSession();
    }

}
```

The environment provides access to the current request (via `getRequest`) and the RpcServlet instance that is handling the current call (via `getRpcServlet`).

Advanced Java topics

Automatic client configuration

The Java RPC backend contains an auto-config mechanism, mainly used for automatically detecting the server URL. You can access it by including the following script tag in your HTML page:

```
<html>
    <head>
        <!-- ... -->
        <script type="text/javascript" src=".qxrpc"></script>
    </head>
</html>
```

Provided the HTML page is part of the webapp (and not loaded via file:*...), and provided that you didn't change the default mapping of the RpcServlet (`.qxrpc`), any request to `http://server/app/foo/bar.qxrpc` (or anything else that ends with .qxrpc) will always be directed to the RpcServlet. The RpcServlet fills a structure with basic information about the server. It may answer with something like

```
qx.core.ServerSettings = {serverPathPrefix: 'http://server/app', ...}
```

and this is used by the `makeServerURL()` helper method in the RPC class. You can use this when instantiating an RPC instance:

```
var rpc = new qx.io.remote.Rpc(
    qx.io.remote.Rpc.makeServerURL(),
    "my.package.MyService"
);
```

This way, you don't need to hardcode the URL of the service. Your client code will work without modifications, no matter what the name of your application is or where it is deployed. By generating absolute URLs you don't have to worry about moving around web pages and scripts in the directory structure, which is a common shortcoming of relative URLs. The auto-configration feature is also convenient if you need to embed a session id into the URL.

Subclassing RpcServlet

It can be useful to create your own version of qooxdoo's `RpcServlet`. Some of the benefits of subclassing it are:

1. **Custom object conversion**: By creating your own subclass, you can provide code for custom conversion of objects. This is especially useful for classes that don't have a default constructor.
2. **Detailed server logging**: You can hook your own code into the method calling mechanism, e.g. to provide detailed failure logging (the JavaScript side only receives rather generic errors).
3. **Property filtering**: For methods that return JavaBeans, you can filter the properties that should be sent to the client. This can save a lot of bandwidth without having to completely wrap the result in a custom object.
4. **Class hinting**: For security reasons, the class hinting mechanism isn't active by default (otherwise, client code could instantiate arbitrary server classes). By overriding a method, you can enable it on a case-by-case basis.

The following example code shows how all of this can be done:

```
package my.package;

import java.lang.reflect.InvocationTargetException;
import java.util.Calendar;
import java.util.Map;

import net.sf.qooxdoo.rpc.RpcServlet;
import net.sf.qooxdoo.rpc.RemoteCallUtils;

import org.json.JSONArray;

public class MyRpcServlet extends RpcServlet {

    protected RemoteCallUtils getRemoteCallUtils() {
        return new RemoteCallUtils() {

            // log exceptions by overriding callCompatibleMethod

            protected Object callCompatibleMethod(Object instance,
                    String methodName, JSONArray parameters)
                    throws Exception {
                try {
                    return super.callCompatibleMethod(instance, methodName, parameters);
                } catch (Exception exc) {
                    exc.printStackTrace();
                    throw exc;
                }
            }

            // influence object conversion

            public Object toJava(Object obj, Class targetType) {
                // insert custom conversion to Java here
                // (default: call super method)
                return super.toJava(obj, targetType);
            }

            public Object fromJava(Object obj)
                throws IllegalAccessException, InvocationTargetException,
                NoSuchMethodException {

                // use Dates instead of Calendars (so that the
```

```
                // client code receives native JavaScript dates)
                if (obj instanceof Calendar) {
                    return super.fromJava(((Calendar) obj).getTime());
                }

                return super.fromJava(obj);
            }

            // filter unwanted bean properties

            protected Map filter(Object obj, Map map) {
                if (obj instanceof Date) {
                    map.remove("timezoneOffset");
                }
                return super.filter(obj, map);
            }

            // class hinting

            protected Class resolveClassHint(String requestedTypeName,
                    Class targetType) throws Exception {
                // allow class hinting in some cases
                // (useful for methods that expect a superclass
                // of SubClassA and SubClassB)
                if (requestedTypeName.equals("my.package.SubClassA") ||
                    requestedTypeName.equals("my.package.SubClassB")) {
                    return Class.forName(requestedTypeName);
                } else {
                    return super.resolveClassHint(requestedTypeName, targetType);
                }
            }
        };
    }
}
```

To make use of class hinting on the client side, you have to send objects with a `class` attribute:

```
rpc.callAsync(handler, "testMethod",
    {"class": "my.package.SubClassA",
     property1: 123,
     property2: 456,
     /* ... */
    });
```

Please note that `class` is a reserved word in JavaScript, so you have to enclose it in quotes.

6.2.2 RPC Servers

Java RPC

Note: This information refers to releases up to 0.7.x. It needs to be updated for the current releases 1.3.1.

Building a qooxdoo test application

The Java backend comes with a `build.xml` file that generates a web application archive (WAR). (In order to use this build file, you need to have Ant installed.) The resulting WAR contains all the necessary server-side and client-side classes to experiment with the RPC mechanism.

To build the test WAR, simply follow these steps on the command line:

```
cd /qooxdoo/frontend
make build
cd /qooxdoo/backend/java
ant
```

Now deploy the WAR in a Java web server of your choice (e.g. Apache Tomcat). You can then point your browser to one of the RPC test pages (e.g. `http://localhost:8080/qooxdoo/sample/html/test/RPC_1.html`) to see the RPC mechanism in action.

Future qooxdoo releases may also include a pre-built Java backend.

Building your own applications

You can use the supplied `build.xml` as a starting point for your own applications. For example, you can modify it to include your own applications instead of the qooxdoo examples. Or you can modify it to build a JAR with the qooxdoo RPC classes and add that to an already existing webapp of yours. In this case, you have to add a mapping for the RpcServlet in your web.xml (see `webapp/WEB-INF/web.xml` in the Java backend).

For development, you can use cross-domain calls (see below). This way, you can load HTML and script files via `file://` URLs, and only the server part needs to be packaged in a WAR. To see any client-side changes, simply reload the page. When you're ready to put the application into production, set cross-domain to false and add the client part to the WAR. There are also more sophisticated solutions (e.g. using a servlet and a custom classloader to load scripts), but these are beyond the scope of this article.

PHP RPC

qooxdoo includes an advanced RPC mechanism for direct calls to server-side methods. It allows you to write true client/server applications without having to worry about the communication details.

As described in the *RPC overview*, qooxdoo RPC is based on JSON-RPC as the serialization and method call protocol. This page describes how to set up and implement a PHP-based server.

Setup

Note: The following information is from the README.CONFIGURE file of the RpcPhp contribution.

The simplest configuration of the PHP JSON-RPC server requires these steps:

- Copy the services directory to the root of your web server's data directory, e.g. `/var/www`
- Ensure that PHP is properly configured. Try placing a file in the services directory called `test.php` which contains this data:

  ```
  <?php
    phpinfo();
  ?>
  ```

  You should then be able to access `http://your.domain.com/services/test.php` and see the `phpinfo()` output. If not, you have a web server / php configuration problem to work out.
- Configure your web server to load index.php if it's found in a directory specified by the URL. By default, the web server probably looks only for `index.html` and `index.htm`, but you want it also to look for `index.php`.

Example

Please see RpcExample for an example of how to use an RPC backend.

To set up the RPC Example application:

1. Change directory into qooxdoo-contrib/trunk/RpcExample
2. Edit config.json such that the "path" inside of the "include" key properly points to the framework's application.json file, and the QOOXDOO_PATH variable inside of the "let" key properly points to the root of the framework source tree.
3. Type "generate.py build". That will create a "build" qooxdoo-contrib/trunk/RpcExample/build directory
4. Copy the contents of the build directory to the web server root directory called "test", so your root contains 'test' and 'services'. If you're on Linux, the best way is with rsync:

```
# mkdir /var/www/test
# rsync -av ./build/ /var/www/test/
```

Note the slash after 'build' so that it copies the **contents** of 'build' into 'test', but not the directory 'build' itself.

1. Browse to `http://your.domain.com/test/index.html` and ensure that the echo test in the first tab runs, and then try the Rpc Server Functionality (async) test in the fourth tab.

RPC with a Perl server

qooxdoo includes an advanced RPC mechanism for direct calls to server-side methods. It allows you to write true client/server applications without having to worry about the communication details.

As described in the *RPC overview*, qooxdoo RPC is based on JSON-RPC as the serialization and method call protocol. This page describes how to set up and implement a Perl-based server.

Setup

Get a copy of the qooxdoo perl backend (`Qooxdoo::JSONRPC`) from our sourceforge download area. In the archive you will find a `README.txt` file as well as a `README.CONFIGURE` which contains details of how to set up the server. The steps involved are:

- First, make sure that you have the Perl `JSON` module installed. This can be found on CPAN, and if you can't get it prepackaged, can be installed with

  ```
  # perl -MCPAN -e 'install JSON'
  ```

- If you care for performance at all, you may want to make sure that you have the `FCGI` module installed as well as mod_fcgid in your apache server.

  ```
  # perl -MCPAN -e 'install FCGI'
  ```

- The `JSONRPC` module requires a module to take care of the session handling. You can either use the `SessionLite` module included with RpcPerl or you can get `CGI::Session` from CPAN.
- Next you'll need to configure a list of places to look for modules and services: Open `jsonrpc.pl` and add as many (space-separated) directories as you need to the lib list. Usually this need only contain the full path to wherever you have put your `Qxoodoo::JSONRPC` module.

  It does however mean that services can be spread across different directories for different projects. These are searched for as `<path>/Qooxdoo/Services/<service name>`, and should have package names such as `Qooxdoo::Services::qooxdoo::test` (which corresponds to `<path>/Qooxdoo/Services/qooxdoo/test.pm`).

The harness will obviously be run as the user that the web server is configured to run as, so needs access to the perl backend files.

- Test that the script has all its dependencies, and can find the runtime module:

```
$ ./jsonrpc.pl
Content-Type: text/html; charset=ISO-8859-1

Your HTTP Client is not using the JSON-RPC protocol
```

If you get "Can't locate Qooxdoo/JSONRPC.pm in @INC" then you didn't get you library path right.

- Now you have a few choices, depending on how you plan to integrate with your web server. The quickest way to get going is to simply copy `jsonrpc.pl` into you `cgi-bin` directory.
- You can now point your web browser at the following address, and confirm that you get the JSON-RPC protocol error shown above.

```
http://localhost/cgi-bin/jsonrpc.pl
```

Writing your own services

Let's start by writing our own first service which will add its arguments up. The service will be called example.wiki and have a method called add.

To do this, we create a package called Qooxdoo::Services::example::wiki which will live in a file `Qooxdoo/Services/example/wiki.pm`. It doesn't matter where this file lives, but it will be searched for using the path(s) that you specified in the jsonrpc.pl harness. For this example you can create the new file under `backend/perl/Qooxdoo/Services/example`. Our service contains:

```
package Qooxdoo::Services::example::wiki;

use strict;

use Qooxdoo::JSONRPC;

sub method_add
{
    my $error  = shift;
    my @params = @_;

    my $count  = 1+$#params;

    if ($count != 1)
    {
        $error->set_error (Qooxdoo::JSONRPC::JsonRpcError_ParameterMismatch,
                           "Expected 1 parameter, but got $count");
        return $error;
    }

    my @numbers = split (/\s+/, $params[0]);

    my $total = 0;
    $total += $_ foreach (@numbers);
```

```
    return $total;
}

1;
```

The service is just a Perl package containing functions called method_* which are exposed through RPC. When called, the first argument will always be an error object, and subsequent ones will be supplied by the calling Javascript. In this example we just add the numbers in the first argument, which is space separated. [In practice we would probably pass each number as a separate argument, but doing it this way allows us to use `RPC_1.html` for testing]

You can also see how the method has done a check on the supplied parameters, and raised an exception which will be raised in the client.

Now, let's give it a try using the `RPC_1.html` test harness. Change the URL to be the address of jsonrpc.pl, for example /cgi-bin/jsonrpc.pl, the service to be example.wiki and the method to be add. Finally, supply a list of numbers in the final field and click 'Send to server' to see a result.

If you get an error, particularly a server error, have a look in Apache's error_log to see if there is an error recorded. There is also a debug flag in `JSONRPC.pm` which can be enabled. All being well, you should receive a popup with the result.

A more advanced example

Let's write something that's a little more "real world" – an address book! We'll use the `NDBM` database backend as I believe you should have it with Perl. We'll provide a couple of helper functions which open and close the database, as well as methods which list the database keys, fetch a record and store a record. These routines can be added to `wiki.pm`.

```
use Fcntl;
use NDBM_File;

use vars qw(%database);

sub open_database
{
    # Please choose a better database path on a public system
    tie %database, 'NDBM_File', '/tmp/database', O_RDWR|O_CREAT, 0666;
}

sub close_database
{
    untie %database;
}

sub method_get_record_ids
{
    my $error  = shift;

    open_database ();
    my @k = keys %database;
    close_database ();

    return \@k;
}

sub method_get_record
{
```

```
    my $error = shift;
    my $id    = shift;

    open_database ();
    my $record = $database{$id};
    close_database ();

    return $record;
}

sub method_set_record
{
    my $error  = shift;
    my $id     = shift;
    my $record = shift;

    open_database ();
    $database{$id} = $record;
    close_database ();

    return $record;
}
```

Now to implement the front-end. Bear with me for a mo while I write it....

RPC with a Python server

*qooxdoo includes an advanced RPC mechanism for direct calls to server-side methods. It allows you to write true client/server applications without having to worry about the communication details.**

As described in the *RPC overview*, qooxdoo RPC is based on JSON-RPC as the serialization and method call protocol. This page describes how to set up and implement a Python-based server.

Setup

Python 2.4 or 2.5 is required to run a JSON-RPC server. Download and install python from python.org if you don't have it.

The JSON-RPC backend itself is based on the *qxjsonrpc* library. The qxjsonrpc package is part of the RpcPython contribution which has links to the software itself. Extract the archive and run python setup.py install as usual. This will install the qxjsonrpc package into the site-packages subdirectory of your python installation.

The backend requires python-cjson or simplejson (also just `import json as simplejson` for Python 2.6 or higher) for JSON serialization. Install at least one of them. Building python-cjson requires a C compiler, but it is much faster (10-100x) than the pure python simplejson package.

A backend based on qxjsonrpc can be run as

- a standalone HTTP server using the qxjsonrpc.http.HTTPServer class or
- a WSGI application using the qxjsonrpc.wsgi.WSGIApplication class.

You can find examples in the downloaded qxjsonrpc archive.

NOTE: The qxjsonrpc package is very young, it should not be used in production. Bug reports are always welcome. Send your reports to Viktor Ferenczi. Thank you in advance.

Writing your own services

Let's start by writing our own first service which will add its arguments up. The service will be called example.wiki and have a method called add.

To do this, we create a service class called ExampleWikiService which will live in a file wiki.py. This file can be created anywhere with the following contents, preserving the indentation of the source code:

```
#!/usr/bin/python
# -*- coding: ascii -*-

import qxjsonrpc
import qxjsonrpc.http

class ExampleWikiService(object):
    def __init__(self):
        self.total=0
    @qxjsonrpc.public
    def add(self, *args):
        for value in args:
            self.total+=value
        return self.total

def main():
    server=qxjsonrpc.http.HTTPServer()
    server.setService('example.wiki', ExampleWikiService())
    server.serve_forever()

if __name__=='__main__': main()
```

Note the `@qxjsonrpc.public` decorator in the service class. The public decorator makes the decorated method accessible to everyone. Undecorated methods are not accessible by RPC clients.

The service is just an executable Python script. Running `wiki.py` will run a JSON-RPC server at localhost:8000 by default. Open the following link in a new browser window or tab:

http://localhost:8000/?id=1&service=example.wiki&method=add&params=[2]

It should show you the JSON-RPC response. The result will be the accumulated total value. Pressing the refresh (`F5`) key will increment the total value by two as passed in the only argument in the params array.

The RPC call was actually made using the HTTP GET method. You can achieve the same result by sending the same request arguments using the HTTP POST or the ScriptTransport protocol. The later is used by the qooxdoo library for cross-domain requests. Use qooxdoo's RPC functionality for best results.

You can change the arguments to be passed to the method by altering the params array in the address bar. Multiple numbers or even floating point values can be added. If you does not add params at all the total won't change.

A more advanced example

To be done.

Using sessions

@qxjsonrpc.session

To be done.

Running as part of a WEB server

- WSGI: Apache 2.0 and mod_wsgi

To be done.

RPC Server Writer Guide

Writing a new JSON-RPC server for use with qooxdoo is fairly easy. If you follow these rules, you should end up with a conformant implementation. See also the other available qooxdoo RPC servers.

JSON

With the exception of the formatting of Javascript `Date` objects, all communication between client and server is formated as JSON, as described and documented at http://json.org.

Date Objects Date objects are a problem in standard JSON encoding, because there is no "literal" syntax for a date in Javascript. In Javascript, nearly everthing can be represented in literal form: objects by `{ ... }`; arrays by `[ ... ]`; etc. The only native type which can not be represented as a literal is a Date. For this reason, a format for passing Dates in JSON is defined here so that all conforming servers can parse the data received from clients.

Date objects are sent as the following 'tokens'.

- The string `new Date(Date.UTC(`
- The **year**, integer, e.g. `2006`
- A comma
- The **month**, 0-relative integer, e.g. `5` is June
- A comma
- The **day** of the month, integer, range: `1-31`
- A comma
- The **hour** of the day on a 24-hour clock, integer, range: `0-23`
- A comma
- The **minute** of the hour, integer, range: `0-59`
- A comma
- The **second** within the minute, integer, range: `0-59`
- A comma
- The **milliseconds** within the second, integer, range: `0-999`
- The string `))`

A resulting Date representation might therefore be:

```
new Date(Date.UTC(2006,5,20,22,18,42,223))
```

Whitespace

- when generating these date strings, implementations SHOULD NOT add white space before/after/between any of the fields within the date string
- when parsing these date strings, implementations SHOULD allow white space before/after/between any of the fields within the date string

Numbers

- when generating these date strings, implementations MUST NOT add leading zeros to the numeric values in the date string. Doing so will cause them to be parsed as octal values. Numbers MUST be passed in decimal (base 10) notation without leading zeros.
- when parsing these date strings, implementations MUST take the integer value of numeric portions of the string as base 10 values, even if leading zeros appear in the string representation of the numbers..

Within the JSON protocol and in JSON messages between peers, `Date` objects are always passed as UTC.

RPC

Remote procedure calls are issued using JSON seralization. The basis for the objects used to send requests and responses are described and defined at http://json-rpc.org, specifically http://json-rpc.org/wiki/specification. This document introduces a number of differences to that specification, based on real-life implementation discoveries and needs. This portion of this document is an edited version of the JSON-RPC specification.

request (method invocation) A remote method is invoked by sending a request to a remote service. The request is a single object serialized using JSON.

It has four properties:

- `service` - A String containing the name of the service. The server may use this to locate a set of related methods, all contained within the specified service. The format of the supported service strings is up to the server implementation.
- `method` - A String containing the name of the method to be invoked. The method must exist within the specified service. The format of the method string is up to the server implementation.
- `params` - An Array of objects to pass as arguments to the method.
- `id` - The request id. This can be of any type. It is used to match the response with the request that it is replying to. (qooxdoo always sends an integer value for id.)

response When the method invocation completes, the service must reply with a response. The response is a single object serialized using JSON.

It has three properties:

- `result` - The Object that was returned by the invoked method. This must be `null` in case there was an error invoking the method.
- `error` - An *Error Object* if there was an error invoking the method. It must be `null` if there was no error. Note that determination of whether an error occurred is based on this property being `null`, not on result being `null`. It is perfectly legal for both to be `null`, indicating a valid result with value `null`.
- `id` - This must be the same id as the request it is responding to.

The Error Object

An error object contains two properties, `origin` and `code`:

origin `origin` - An error can be originated in four locations, during the process of initiating and processing a remote procedure call. Each possible origin is assigned an integer value, assigned to this property, as follows:

- `1` = the server can return errors to the client
- `2` = methods invoked by the server can return errors
- `3` = Transport (e.g. HTTP) errors can occur
- `4` = the client determined that an error occurred, e.g. timeout

A conforming server implementation MUST send value `1` or `2` and MAY NOT send any other value, for origin. A client may detect Transport or locally-ascertained errors, but a server will never return those.

code `code` - An integer error code. The value of code is hierarchically linked to origin; e.g. the same code represents different errors depending on the value of origin.

One of these values of code SHALL be sent if origin = `1`, i.e. if the server detected the error.

- Error code, value `1`: Illegal Service The service name contains illegal characters or is otherwise deemed unacceptable to the JSON-RPC server.
- Error code, value `2`: Service Not Found The requested service does not exist at the JSON-RPC server.
- Error code, value `3`: Class Not Found If the JSON-RPC server divides service methods into subsets (classes), this indicates that the specified class was not found. This is slightly more detailed than "Method Not Found", but that error would always also be legal (and true) whenever this one is returned.
- Error code, value `4`: Method Not Found The method specified in the request is not found in the requested service.
- Error code, value `5`: Parameter Mismatch If a method discovers that the parameters (arguments) provided to it do not match the requisite types for the method's parameters, it should return this error code to indicate so to the caller.
- Error code, value `6`: Permission Denied A JSON-RPC service provider can require authentication, and that authentication can be implemented such the method takes authentication parameters, or such that a method or class of methods requires prior authentication. If the caller has not properly authenticated to use the requested method, this error code is returned.

If origin = `2`, i.e. the application (invoked method) detected the error, the value of the code property is entirely by agreement between the invoking client and the and invoked method.

message `message` - A free-form textual message describing the error.

Other Errors

Errors detected by the server which indicate that the received data is not a JSON-RPC request SHOULD be simple text strings returned to the invoker, describing the error. A web browser user who accidentally hits the URL of a JSON-RPC server should receive a textual, not Error Object, response, indicating that the server is expecting a JSON-RPC request.

Transport

There are exactly two standard transport facilities potentially used by qooxdoo's qx.io.remote.Rpc class:

- XmlHTTPRequest : The parameters of the remote procedure call are passed to the server using XmlHTTPRequest. The request will be issued using the `POST` method with `Content Type: application/json`. The data provided by the client will be the JSON-serialized request object. The JSON-serialized result object MUST be returned with `Content Type: application/json`. This transport will be used unless the request is issued as cross-domain.
- Script : If the client application invokes a cross-domain request, the request will be issued by URL-encoding the request object and wrapping it in a `<script>` tag. The request uses the `GET` method with `Content Type: text/javascript`. The response to a request received via this method MUST be a call to the static method `qx.io.remote.transport.Script._requestFinished` with parameters of the script id (a copy of the value of the incoming parameter `_ScriptTransport_id`) and the JSON-serialized result object.

A server SHOULD issue an `Other Error` (textual reply) if it detects a method / content type pair other than the two supported ones.

Testing A New Server

To validate that your new server is operating properly, the following test methods may be created at your server:

- `echo` - Echo the one and only parameter back to the client, in the form: `Client said: [ <parameter> ]` where all text is literal except for `<parameter>`.
- `sink` - Sink all data and never return. ("Never" is a long time, so it may be simulated by sleeping for 240 seconds.
- `sleep` - Sleep for the number of seconds provided as the first parameter, and then return that parameter.
- `getInteger` - Return the integer value `1`
- `getFloat` - Return the floating point value `1/3`
- `getString` - Return the string `"Hello world"`
- `getArrayInteger` - Return an array containing the four integers `[ 1, 2, 3, 4 ]` in that order.
- `getArrayString` - Return an array containing the four strings `[ "one", "two", "three", "four" ]` in that order
- `getObject` - Return some arbitrary object
- `getTrue` - Return the binary value `true`
- `getFalse` - Return the binary value `false`
- `getNull` - Return the value `null`
- `isInteger` - Return `true` if the first parameter is an integer; `false` otherwise
- `isFloat` - Return `true` if the first parameter is a float; `false` otherwise
- `isString` - Return `true` if the first parameter is a string; `false` otherwise
- `isBoolean` - Return `true` if the first parameter is either one of the boolean values `true` or `false`; return `false` otherwise.
- `isArray` - Return `true` if the first parameter is an array; `false` otherwise
- `isObject` - Return `true` if the first parameter is an object; `false` otherwise

- isNull - Return true if the first parameter is the value null; false otherwise.
- getParams - Echo all parameters back to the client, in received order
- getParam - Echo the first parameter back to the client. This is a synonym for the echo method.
- getCurrentTimestamp - Return an object which has two properties:
  - now: An integer representing the current time in a native format, e.g. as a number of seconds or milliseconds since midnight on 1 Jan 1970.
  - json: A *Date* object representing that same point in time

A test of all of the primitive RPC operations is available in the qooxdoo-contrib project RpcExample. The third tab provides a test of the operations using synchronous requests, and the fourth tab tests the operations using asyncronous requests. Note that the results are displayed in the debug log (in Firebug or in the debug console enabled by pressing F7). You can look for true as a result of each request.

A future test will validate that all returned values are as expected, and display a single "passed/fail" indication.

6.3 Specific Widget Communication

6.3.1 Using the remote table model

The remote table should be used whenever you want to display a large amount of data in a performant way.

As this table model loads its data on-demand from a backend, only those rows are loaded that are near the area the user is currently viewing. If the user scrolls, the rows that will be displayed soon are loaded asynchronously in the background. All loaded data is managed in a cache that automatically removes the last recently used rows when it gets full.

To get this model up and running you have to implement the actual loading of the row data by yourself in a subclass.

Implement your subclass

To correctly implement the remote table model you have to define/overwrite two methods _loadRowCount and _loadRowData. Both are automatically called by the table widget.

```
qx.Class.define("myApplication.table.RemoteDataModel",
{
  extend : qx.ui.table.model.Remote,

  members :
  {
     // overloaded - called whenever the table requests the row count
    _loadRowCount : function()
    {
      // Call the backend service (example) - using XmlHttp
      var url  = "http://localhost/services/getTableCount.php";
      var req = new qx.io.remote.Request(url, "GET", "application/json");

      // Add listener
      req.addListener("completed", this._onRowCountCompleted, this);

      // send request
      req.send();
    },
```

```
    // Listener for request of "_loadRowCount" method
    _onRowCountCompleted : function(response)
    {
       var result = response.getContent();
       if (result != null)
       {
          // Apply it to the model - the method "_onRowCountLoaded" has to be called
          this._onRowCountLoaded(result);
       }
    },

    // overloaded - called whenever the table requests new data
    _loadRowData : function(firstRow, lastRow)
    {
       // Call the backend service (example) - using XmlHttp
       var baseUrl  = "http://localhost/services/getTableRowData.php";
       var parameters = "?from=" + firstRow + "&to=" + lastRow;
       var url = baseUrl + parameters;
       var req = new qx.io.remote.Request(url, "GET", "application/json");

       // Add listener
       req.addListener("completed", this._onLoadRowDataCompleted, this);

       // send request
       req.send();
    },

     // Listener for request of "_loadRowData" method
    _onLoadRowDataCompleted : function(response)
    {
        var result = response.getContent();
       if (result != null)
       {
          // Apply it to the model - the method "_onRowDataLoaded" has to be called
          this._onRowDataLoaded(result);
       }
    }
  }
});
```

Using your remote model

Now that you've set up the remote table model the table component can use it.

```
var remoteTableModelInstance = new myApplication.table.RemoteDataModel();
yourTableInstance.setTableModel(remoteTableModelInstance);
```

That's all you need to ensure your table is using your remote model.

Sorting your data

The table component offers sortable columns to let users sort the data the way they like. You can enable this sorting ability for each column. Since you have to pull the data into the table yourself you have to update the table data once the user changes the sorting criteria. You have to enhance the `_loadRowData` method with this information to inform your backend how to sort the data.

```
// "_loadRowData" with sorting support
_loadRowData : function(firstRow, lastRow)
{
    // Call the backend service (example) - using XmlHttp
    var baseUrl  = "http://localhost/services/getTableRowData.php";
    var parameters = "?from=" + firstRow + "&to=" + lastRow;

    // get the column index to sort and the order
    var sortIndex = this.getSortColumnIndex();
    var sortOrder =  this.isSortAscending() ? "asc" : "desc";

    // setting the sort parameters - assuming the backend knows these
    parameters += "&sortOrder=" + sortOrder + "&sortIndex=" + sortIndex;

    var url = baseUrl + parameters;
    var req = new qx.io.remote.Request(url, "GET", "application/json");

    // Add listener
    req.addListener("completed", this._onLoadRowDataCompleted, this);

    // send request
    req.send();
}
```

Backend

The backend has to deliver the requested data in a JSON data structure in order to display the data correctly. The data structure has to use the same IDs as the remote table model instance at the client-side.

For example

```
var remoteModel = new myApplication.table.RemoteDataModel();

// first param: displayed names, second param: IDs
remoteModel.setColumns( [ "First name", "Last name" ], [ "first", "last" ] );
```

Then the data delivered by the backend should have the following structure:

```
result = {[
  { "first" : "John", "last" : "Doe"  },
  { "first" : "Homer", "last" : "Simpson"  },
  { "first" : "Charlie", "last" : "Brown"  },
  ...
]};
```

Moreover, the backend has to deliver the row count, i. e. the number of rows the table contains. This is what the `_loadRowCount` function of your subclass expects to get. Please make sure that the URLs `http://localhost/services/getTableCount.php` and `http://localhost/services/getTableRowData.php` of your subclass point to the right location.

Summary

This short and very basic example is far from complete and in your application you will have to implement some more features like error-handling, but it should give you a short overview of how to implement the remote table model in qooxdoo.

Another basic implementation which uses the PHP RPC backend is available in qooxdoo-contrib. Take a look at the RPCExample and setup the necessary RPC PHP backend.

CHAPTER

SEVEN

DEVELOPMENT

7.1 Debugging

7.1.1 Logging System

The logging API allows for a definition of what is logged and where it is logged, while trying to keep usage as simple as possible.

Writing Log Messages

Every qooxdoo object has four logging methods `debug()`, `info()`, `warn()` and `error()`. Each method takes an arbitrary number of parameters which can be of any JavaScript data type: The logging system will create text representations of non-string parameters.

The name of the method defines the log level your log message will get. If you want to log a message that is interesting for debugging, then use `debug()`. If you want to log some general information, use `info()`. If you want to log a warning, use `warn()`. Errors should be logged with `error()`. Have a look to the API documentation of the class `qx.core.Object` for more information.

So to write a log message just call:

```
this.debug("Hello world");
```

Writing Custom Log Appenders

qooxdoo's logging system is extensible by adding user-defined log appenders. These can be used in place of or in addition to qooxdoo's default appenders. A log appender is a static class with at least a "process" method. This method will be called by the logger with an "entry" map as the only parameter. Log appenders that need only a text representation of the logged item(s) can pass this map to `qx.log.appender.Util.toText`. For other use cases, this is what an entry map consists of:

Log Entry Map

- *items* Array of maps containing information about the logged items, see below
- *time* When the message was logged (JavaScript Date)
- *level* The level of the log message
- *object* qx object registry hash of the object the log method was called on

- *win* The application's DOM window (necessary for cross-frame logging)
- *offset* Time in milliseconds since application startup

Logged Item Map

- *text* Text representation of the logged item. If the logged item is an array, the value of *text* is an array containing text representations of each of the logged array's entries. For maps, the value is an array of maps with the following fields:
- *key* The map key's name
- *text* Representation of the corresponding value
- *trace* Stack trace (if the logged item is an Error object)
- *type* One of "undefined", "null", "boolean", "number", "string", "function", "array", "error", "map", "class", "instance", "node", "stringify", "unknown" "stringify"

Registering Appenders

To register an appender named "custom.Logger" with qooxdoo's logging system, simply call

```
qx.log.Logger.register(custom.Logger)
```

7.1.2 Debugging Applications

You have several options at hand when it comes to debugging a qooxdoo application.

Introspection

- `qx.io.Json.stringify()`
- `qx.dev.Debug()`

Included in the latter is qx.dev.Debug.debugObject() which will print out a complete recursive hierarchy (or up to some max level) of an object.

This is taken from a firebug interactive session:

```
>>> var myTest = {a:1, b:[2,3], c:4}
>>> qx.dev.Debug.debugObject(myTest)
1665905: Object, count=3:
------------------------------------------------------------
a: 1
b: Array
 0: 2
 1: 3
c: 4
==================================
```

Memory leaks

- Setting `qx.disposerDebugLevel`

AJAX

- Setting `qx.ioRemoteDebug`
- Setting `qx.ioRemoteDebugData`

Debugging Tools

Some browser-specific tools allow for a powerful and often convenient way of debugging applications.

Code Instrumentation Idioms

These are helpful idioms you might want to include in your code, i.e. you use them at *programming time*.

this.debug()

With `this.debug()` you can print out any string you want to see in the console during execution of your application. Of course you might want to interpolate variable values in the output. If you pass an entire object reference, the whole object will by stringyfied and printed. So beware: for big objects you get the entire instantiation in code printed out!

Example:

```
this.debug("I found this value for myVar: "+myVar);
```

console.log()

In contrast to this.debug(), if you pass an object reference to `console.log()` Firebug will provide you with a nice hyperlink to the live object which you can follow to inspect the object in a structured way. This is much easier to navigate than to skim through pages of source output.

```
var b = new qx.ui.form.Button();
console.log(b);
```

this.trace()

Will log the current stack trace using the defined logger. This can be useful to inspect from which method the current function was called.

```
this.trace()
```

Getting at your Objects

This section shows you how to access objects of your application at *run time*, i.e. while it executes. Access to those objects is possible through JavaScript, either in the form of another piece of JavaScript code, or - especially interesting for debugging - from an interactive shell, like Firebug or Venkman, that allows for interactive input and execution of JavaScript commands.

qx.core.Init.getApplication()

In your running app, the singlton `Init` object provides you with the `getApplication()` method, to access the root object of your application. All members and sub-members that you have attached to your application class in your code are accessible this way.

```
qx.core.Init.getApplication();
```

Firebug Usage Idioms

"Inspect"

Getting at your application objects fast is a common requirement when debugging. A useful idiom (or usage pattern) with Firebug is to press the *"Inspect"* button and select the visible page element you are interested in. This will take Firebug to its HTML tab in a split-pane view. The left side holds the HTML code underlying your selection (which is probably not very enlightening). The right side though has a *"DOM"* tab, among others. Selecting this will show a display of the underlying DOM node, which features a `qx_Widget` attribute. This attribute is added to the outermost HTML tag representing a qooxdoo widget. For complex widgets that are made up of nested HTML elements, make sure to select the outermost container node that actually has this attribute `qx_Widget`. It takes you straight to the qooxdoo object associated with the selected DOM node.

```
Inspect -> Web page element -> HTML tab -> right side -> DOM tab -> qx_Widget link
```

7.2 Performance

7.2.1 Memory Management

Introduction

Generally, qooxdoo's runtime will take care of most of the issues around object disposal, so you don't have to be too anxious if you get those 'missing destruct declaration' messages from a verbose disposer run.

To destruct existing objects at the end of your application is an important feature in the ever growing area of web applications. Widgets and models are normally handling a few storage fields on each instance. These fields need the dispose process to work without memory leaks.

Normally, JavaScript automatically cleans up. There is a built-in garbage collector in all engines. But these engines are more or less buggy. One problematic issue is that browsers differentiate between DOM and JavaScript and use different garbage collection systems for each (This does not affect all browsers, though). Problems arise when objects create links between the two systems. Another issue are circular references which could not be easily resolved, especially by engines which rely on a reference counter.

To help the buggy engines to collect the memory correctly it is helpful to dereference complex objects from each other, e.g. instances from maps, arrays and other instances. You don't need to delete primitive types like strings, booleans and numbers.

qooxdoo has solved this issue from the beginning using the included "dispose" methods which could be overridden and extended by each class. qooxdoo 0.7 introduced a new class declaration. This class declaration supports real "destructors" as known from other languages. These destructors are part of the class declaration. The new style makes it easier to write custom destructor/disposer methods because there are many new helper methods and the whole process has been streamlined to a great extend.

Disposing an application

You can dispose any qooxdoo based application by simply calling `qx.core.ObjectRegistry.shutdown()`. The simplest possibility is to use the command line included in Firebug. Another possibility is to add a HTML link or a button to your application which executes this command.

You can look at the dispose behaviour of your app if you set the disposer into a verbose mode and then invoke it deliberately while your app is running. This will usually render your app unusable, but you will get all those messages hinting you at object properties that might need to be looked after. How-To instructions can be found *here*. But mind that the disposer output contains only hints, that still need human interpretation.

Example destructor

```
destruct : function()
{
  this._data = this._moreData = null;
  this._disposeObjects("_buttonOk", "_buttonCancel");
  this._disposeArray("_children");
  this._disposeMap("_registry");
}
```

- `_disposeObjects`: Supports multiple arguments. Dispose the objects (qooxdoo objects) under each key and finally delete the key from the instance.
- `_disposeArray`: Disposes the array under the given key, but disposes all entries in this array first. It must contain instances of qx.core.Object only.
- `_disposeMap`: Disposes the map under the given key, but disposes all entries in this map first. It must contain instances of qx.core.Object only.

How to test the destructor

The destructor code allows you an in-depth analysis of the destructors and finds fields which may leak etc. The DOM tree gets also queried for back-references to qooxdoo instances. These checks are not enabled by default because of the time they need on each unload of a typical qooxdoo based application.

To enable these checks you need to select a variant and configure a setting.

The variant `qx.debug` must be `on`. The setting `qx.disposerDebugLevel` must be at least at `1` to show not disposed qooxdoo objects if they need to be deleted. A setting of `2` will additionally show non qooxdoo objects. Higher values mean more output. Don't be alarmed if some qooxdoo internal showing up. Usually there is no need to delete all references. Garbage collection can do much for you here. For a general analysis `1` should be enough. You need to add a setting named "qx.disposerDebugLevel" with the value `1` to your `config.json`. See at the Support for finding potential memory leaks snippet how to change your configuration.

Log output from these settings chould look something like this:

```
35443 DEBUG: testgui.Report[1004]: Disposing: [object testgui.Report]FireBug.js (line 75)
Missing destruct definition for '_scroller' in qx.ui.table.pane.FocusIndicator[1111]: [object qx.ui.t
Missing destruct definition for '_lastMouseDownCell' in qx.ui.table.pane.Scroller[1083]: [object Obje
036394 DEBUG: testgui.Form[3306]: Disposing: [object testgui.Form]FireBug.js (line 75)
Missing destruct definition for '_dateFormat' in qx.ui.component.DateChooserButton[3579]: [object qx
Missing destruct definition for '_dateFormat' in qx.ui.component.DateChooserButton[3666]: [object qx
```

The nice thing here is that the log messages already indicate which dispose method to use: Every *"Missing destruct..."* line contains a hint to the type of member that is not being disposed properly, in the *"[object ...]"* part of the line. As a rule of thumb

- native Javascript types (Number, String, Object, ...) usualy don't need to be disposed.
- for qooxdoo objects (e.g. qx.util.format.DateFormat, testgui.Report, ...) use `_disposeObjects`
- for arrays or maps of qooxdoo objects use `_disposeArray` or `_disposeMap`.
- be sure to cut all references to the DOM because garbage collection can not dispose object still connected to the DOM. This is also true for event listeners for example.

7.2.2 Profiling Applications

qooxdoo has built-in a cross-browser, pure JavaScript profiler. If the profiler is enabled, each call of a method defined by qooxdoo's class declaration can be measured. The profiler is able to compute both the total own time and the call count of any method.

Since the profiler is implemented in pure JavaScript, it is totally cross-browser and works on any supported browser.

How to enable the Profiler

Basically set the variant `qx.aspects` to `on` and be sure to include the class qx.dev.Profile. The class should be included before other classes. The easiest way to achieve that is to extend the `profiling` helper job in a job that creates your application. For example, to enable profiling in the source version of your app, go to the `"jobs"` section of your config.json, and add

```
"source-script" : {
  "extend" : [ "profiling" ]
}
```

How to use the Profiler

The profiler can be controlled either hard-wired in the application code, or interactively using a JavaScript shell like FireBug for Firefox or DebugBar for IE.

Profiling a certain action:

- Open the application in your browser
- At the JavaScript console type `qx.dev.Profile.stop()` to clear the current profiling data gathered during startup
- Start profiling using `qx.dev.Profile.start()`
- Perform the action you want to profile
- Stop profiling using `qx.dev.Profile.stop()`
- Open the profiler output window: `qx.dev.Profile.showResults(50)`. The parameter specifies how many items to display. Default value is set to 100. The output will be sorted by the total own time of each method. Alternatively you can work with the raw profiling data returned by `qx.dev.Profile.getProfileData()`.

Limitations

In order to interpret the results correctly it is important to know the limitations of this profiling approach. The most significant limitation is due to the fact that the profiler itself is written in JavaScript and runs in the same context as the application:

- The profiler adds some overhead to each function call. The profiler takes this overhead into account in the calculation of the own time but there can still be a small inaccuracy.
- The result of `new Date()`, which is used for timing, has a granularity of about 10ms on many patforms, so it is hard to measure especially small functions accurately.
- The application is slowed down because profiling is done by wrapping each function. Profiling should always be turned off in production code before deployment.
- Turning on profiling currently breaks most applications in Safari 3.0.2 due to a very limited maximum recursion depth of only 100 (Bugzilla Bug 226). Since the profiler has to wrap each function, the call stack is doubled, which is just too much for Safari.

Summary

The output of the profiler can be of great value to find hot spots and time-consuming code. The timing data should be interpreted rather qualitatively than quantitatively, though, due to constraints of this approach.

Note: The application is slowed down because profiling is done by wrapping each function. Profiling should always be turned off in production code before deployment.

7.3 Testing

7.3.1 Unit Testing

qooxdoo comes with its own, nicely integrated unit testing environment and the corresponding application called Testrunner. While being similar to JSUnit, the solution that ships with the qooxdoo SDK does not require any additional software.

If you look at the component section of a qooxdoo distribution, you will find the Test Runner tailored to test the functionality of the qooxdoo *framework*. It provides a convenient interface to test classes that have been written to that end. You can run single tests, or a whole suite of them at once.

But the Test Runner framework can be deployed for your *own application*. It provides a GUI, a layer of infrastructure and a certain interface for arbitrary test classes. You can write your own test classes and take advantage of the Test Runner environment.

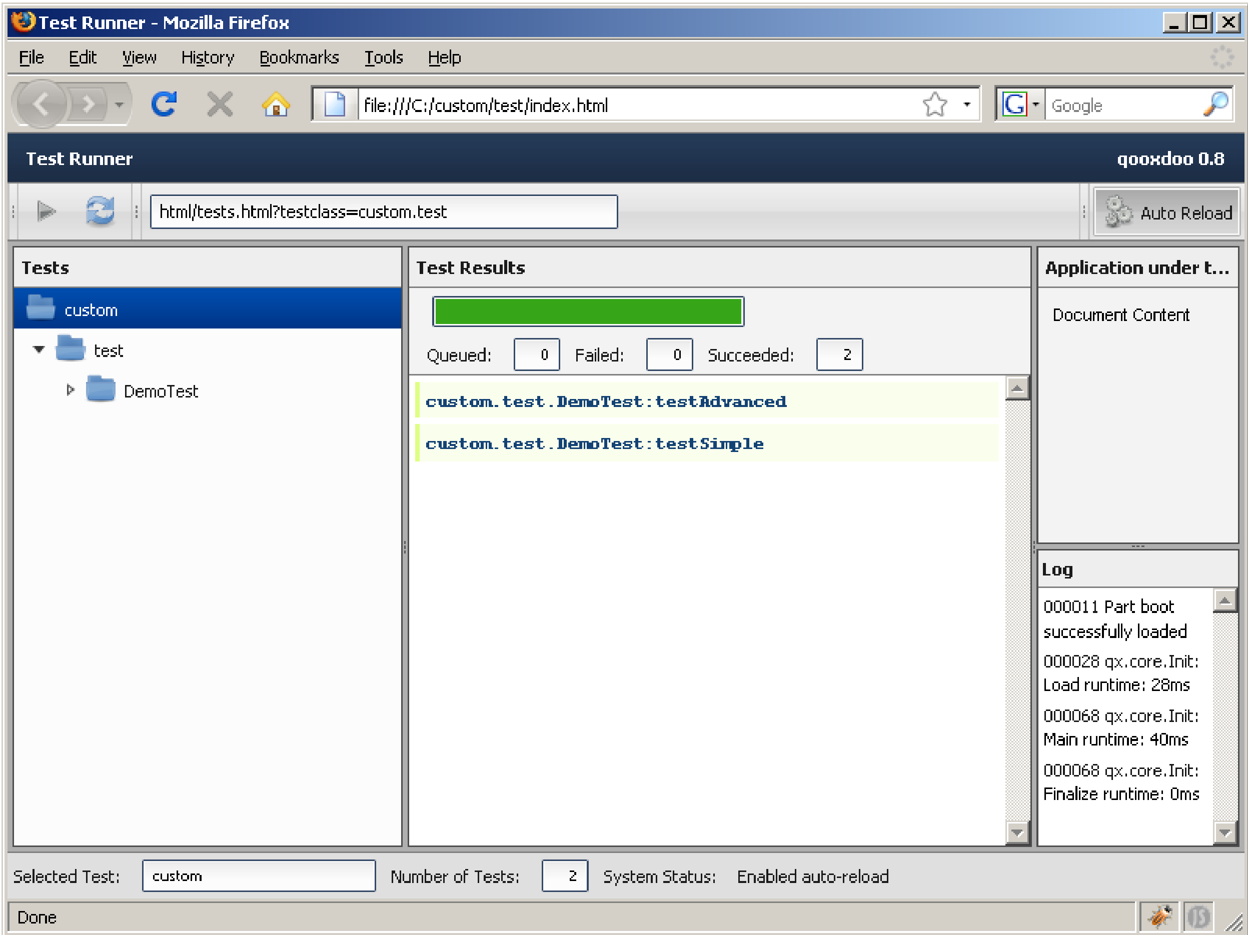
- Test Tools – an overview over test tools and approaches
- *The qooxdoo Test Runner* – how to deploy the Testrunner component for your own application
- *Framework Unit Testing* – a developer's notebook about unit testing of the qooxdoo framework

7.3.2 The qooxdoo Test Runner

"Test Runner" is a unit testing framework that fully supports testing qooxdoo classes. It is similar to but does not require JSUnit or any other JavaScript unit testing framework. If you look at the component section of a qooxdoo distribution under `component/testrunner/`, you will find the Test Runner sources, together with a mockup test class. In the `framework/` section you can create a Test Runner instance with all test classes from the qooxdoo framework by running:

```
./generate.py test
```

Test Runner provides a convenient interface to test classes that have been written to that end. You can run single tests, or run a whole suite of them at once.



Note: See the Test Runner in action in the online demo.

The Test Runner framework can also be deployed for *your own* application. It provides a GUI, a layer of infrastructure and a certain interface for arbitrary test classes. So now you can write your own test classes and take advantage of the Test Runner environment.

How to deploy Test Runner for your own development

This section assumes that your qooxdoo application bears on the structure of the qooxdoo *skeleton* application. Then this is what you have to do:

Writing Test Classes

- You have to code test classes that perform the individual tests. These test classes have to comply to the following constraints:
  - They have to be within the name space of your application.
  - They have to be derived from `qx.dev.unit.TestCase`.
  - They have to define member functions with names starting with `test*`. These methods will be available as individual tests.

- Apart from that you are free to add other member functions, properties etc., and to instantiate other classes to your own content. But you will usually want to instantiate classes of your current application and invoke their methods in the test functions.
- In order to communicate the test results back to the Test Runner framework exceptions are used. No exception means the test went fine, throwing an exception from the test method signals a failure. Return values from the test methods are not evaluated.
- To model your test method behaviour, you can use the methods inherited from `qx.dev.unit.TestCase` which encapsulate exceptions in the form of assertions:
  * `assert`, `assertFalse`, `assertEquals`, `assertNumber`, ... - These functions take values which are compared (either among each other or to some predefined value) and a message string, and raise an exception if the comparison fails.
  * A similar list of methods of the form `assert*DebugOn` is available, which are only evaluated if the debug variant `qx.debug` is on (see *Variants*).
  * See the documentation for the qx.dev.unit.TestCase class for more information on the available assertions.

Generic setUp and tearDown Test classes can optionally define a `setUp` method. This is used to initialize common objects needed by some or all of the tests in the class. Since `setUp` is executed before each test, it helps to ensure that each test function runs in a "clean" environment. Similarly, a method named `tearDown` will be executed after each test, e.g. to dispose any objects created by `setUp` or the test itself.

Specific tearDown For cases where the generic class-wide `tearDown` isn't enough, methods using the naming convention `tearDown<TestFunctionName>` can be defined. A method named e.g. `tearDownTestFoo` would be called after `testFoo` and the generic `tearDown` of the class were executed.

Asynchronous Tests Starting with qooxdoo 0.8.2, the unit testing framework supports asynchronous tests. This enables testing for methods that aren't called directly, such as event handlers:

- Test classes inheriting from `qx.dev.unit.TestCase` have a `wait()` method that stops the test's execution and sets a timeout. `wait()` should always be the last function to be called in a test since any code following it is ignored. `wait()` has two optional arguments: The **amount of time to wait** in milliseconds (defaults to 5000) and a **function to be executed** when the timeout is reached. If no function is specified, reaching the timeout will cause an exception to be thrown and the test to fail.
- The `resume()` method is used to (surprise!) resume a waiting test. It takes two arguments, a **function to be executed** when the test is resumed, typically containing assertions, and the object context it should be executed in.

Here's an example: In our test, we want to send an AJAX request to the local web server, then assert if the response is what we expect it to be.

```
testAjaxRequest : function()
{
  var request = new qx.io.remote.Request("/myWebApp/index.html");
  request.addListener("completed", function (e) {
    this.resume(function() {
      this.assertEquals(200, e.getStatusCode());
    }, this);
  }, this);
  request.send();
```

```
  this.wait(10000);
}
```

Create the Test Application

- Run `generate.py test` from the top-level directory of your application. This will generate the appropriate test application for you, which will be available in the subfolder `test` as `test/index.html`. Open this file in your browser and run your tests.
- Equally, you can invoke `generate.py test-source`. This will generate the test application, but allows you to use the *source* version of your application to run the tests on. In doing so the test application links directly into the source tree of your application. This allows for test-driven development where you simultaneously develop your source classes, the test classes and run the tests. All you need to do is to change the URL of the "test backend application" (the textfield in the upper middle of the TestRunner frame) from `tests.html` (which is the default) to `tests-source.html`. (Caveat: If `generate.py test-source` is the first thing you do, you might get an error when TestRunner starts, since the default tests.html has not been built; just change the URL and continue). For example, the resulting URL will look something like this:

  ```
  html/tests-source.html?testclass=<your_app_name>
  ```

  After that, you just reload the backend application by hitting the reload button to the right to see and test your changes in the TestRunner.
- If you're working on an application based on qx.application.Native or qx.application.Inline (e.g. by starting with an Inline skeleton), you can run `generate.py test-native` or `generate.py test-inline` to create a test application of the same type as your actual application. The TestRunner's index file will be called `index-native.html` or `index-inline.html`, respectively.

7.3.3 Test Runner 2 (Experimental)

As an alternative to the "regular" Test Runner GUI, test applications can be run in the new "testrunner2" component. This is a modular unit testing GUI that makes use of framework features such as data binding that were introduced after the original Test Runner was created. Its main advantage is separation of logic and UI so that specialized views for different use cases can be created, such as a lightweight HTML GUI for use on mobile devices, or a "headless" UI for server-side tests running in Rhino or node.js.

Test Runner 2 is designed to be fully backwards compatible with existing unit test suites. At any time, developers can switch between the old an new runners using the `TESTRUNNER_TYPE` configuration macro. This can be defined in an application's config.json file or on the command line:

```
generate.py test -m TESTRUNNER_TYPE:testrunner2
```

The generated files and directories use the same names as the original Test Runner.

Defining Test Requirements

Test Requirements are a new feature only supported by Test Runner 2. Basically, they are conditions that must be met before a test can be run. For example, a test might rely on the application having been loaded over HTTPS and would give false results otherwise. Requirements are defined for individual tests; if one or more aren't satisfied, the test code won't be executed and the test will be marked as "skipped" in Test Runner 2's results list.

Using Requirements

The make use of the requirements feature, test classes must include the MRequirements mixin. The mixin defines a method `require` that takes an array of strings: The requirement IDs. This method is called from a test function **before** the actual logic of the test, e.g.:

```
testSslRequest : function()
{
  this.require(["ssl"]);
  // test code goes here
}
```

`require` then searches the current test instance for a method that verifies the listed requirements: The naming convention is "has" + the requirement ID with the first letter capitalized, e.g. `hasSsl`. This method is the called with the requirement ID as the only parameter. If it returns `true`, the test code will be executed. Otherwise a RequirementError is thrown. Test Runner 2 will catch these and mark the test as "skipped" in the results list. Any test code after the `require` call will not be executed.

In addition to the verification methods in MRequirements, test developers can define their own right in the test class.

7.3.4 Framework Unit Testing

This page is about creating unit tests for the qooxdoo framework classes. It is a developer's notebook to collect ideas and approaches to create working unit tests and a good test coverage for the framework.

Currently, it is just a list of notes:

- With 1.2 the framework's unit test classes are part of the framework class library, under the `qx.test.*` name space.
- The existing tests cover only a portion of the framework classes.
- The potential to create a unit test for any given framework class largely depends on the level of sophistication the test should have, which condition it tests and what is considered correctness: * on the simplest level, a test could just run through the methods of a class and invoke them; test success is defined by the absence of runtime errors (exceptions etc.). This is sometimes called `smoke testing`. * on more sophisticated levels, correctness can be defined by return values, changes of system state, manipulations of the underlying DOM, up to GUI changes in the browser. Obviously, with each level it's get harder to test and/or check the correctness.
- Currently, we'd rather have a large test coverage with simple tests, than have sophisticated tests for only a few classes. The test sophistication level can then be increased individually step by step.
- The **event** system should be black-box (API) testable, but there are currently only few tests written (?).
- **io.Remote** could be tested with a suitable server backend running in the test environment
- The **DOM/BOM** layer is black-box testable and there is acceptable test coverage
- The **layout** system should be black-box testable.
- The core **Widget** class should be black-box testable.
- Higher-level **GUI widgets** are difficult to black-box test, since they require user and GUI interaction (?). Selenium RC could be used here, but requires additional environment setup.
- Maybe we can come up with a good classification of the framework classes, and how each class can and should be tested.
- Automated GUI tests depend on synthetic events being generated. Selenium can do this. Other possibilities?

- The Testrunner application is a nice GUI tool for interactive testing, but for automated/continuous-integration testing we need a (nearly)non-GUI test frame.

7.4 Parts

7.4.1 Parts and Packages Overview

Note: This is still an experimental feature.

Motivation

Packages are a concept that allows you to partition your application physically. The idea is to spread the entire application over multiple JavaScript files, in order to optimize download and startup behaviour. On page load only the essential first part of the application is loaded (commonly called the *boot* part), while others remain on the server and will only be loaded on demand. As a consequence, the initial code part is smaller, so it's faster to download, consumes less bandwidth and starts up faster in the browser. Other parts are then loaded on demand during the user session. This incures a bit of latency when the user enters a certain application path for the first time and the correpsonding part has to be loaded. On the other side, parts that pertain to a certain application path (e.g. an options dialogue) never have to be downloaded if this application path is not entered during the running session.

Development Model

In order to realize this concept, you have the option to specify *parts* of your application, while the build process takes care of mapping these (logical) parts to physical packages that are eventually written to disk. At run time of your application, the inital package will contain loader logic that knows about the other parts.

There are two different but related terms here: You as the developer define **parts** of your application. These are logical or visual related elements, like all elemens that make up a complex dialogue, or the contents of an interactive tab pane. The build process then figures out all the dependencies of these parts and collects them into **packages**, which eventually map to physical files on disk. Since some parts might have overlapping dependencies, these are optimized so that they are not included twice in different packages. Also, you might want to specify which parts should be collapsed into as few packages as possible, how small a package might be, and so forth. So you define the logical partitioning of your application and specify some further constraints, and the build process will take care of the rest, producing the best physical split of the entire app under the given constraints.

Loading Parts

In your application code, you then load the defined parts at suitable situations, e.g. when the user opens a dialogue defined as a part, using qooxdoo's PartLoader API. The PartLoader keeps track of which parts have already been loaded, and provides some further housekeeping. But it is your responsibility to "draw in" a given part at the right moment.

Consequently, the configuration of your application allows you to specify those logical parts of your application, by giving a suitable name to each and listing the top-level classes or class patterns for each. You are using these part names with the PartLoader in your application code. Further config keys allow you tailor more specifics, as mentioned above. See the *packages key* reference section for the config key nitty-gritty.

7.4.2 Using Parts

Basic Usage

Parts allow you partition your application into multiple Javascript files.There is an initial part, the *boot* part, that is loaded at application start-up. All other parts have to be loaded explicitly in your application code.

To use parts in your application, you have to do two things:

- declare the parts in your application's *config.json* configuration file
- load each part other than the boot part explicitly at suitable situations in your application code

Here is an example:

Suppose you have a settings dialog in your application that is only needed occasionally. You want to save the memory footprint of the involved classes, and only load the dialog on demand when the user hits an "Open Settings Dialog" button during a session. If the user doesn't invoke the dialog, the necessary classes are not loaded and the application uses less memory in the browser. In all cases, application start-up is faster since less code has to be loaded initially.

Add Parts to your Config

In your configuration file, add the following job (assuming you are using a standard GUI application with a name space of *custom*):

```
"build-script":
{
  "packages" :
  {
    "parts"  :
    {
      "boot"     :
      {
        "include" : [ "${QXTHEME}", "custom.Application" ]
      },
      "settings" :
      {
        "include" : [ "custom.Settings" ]
      }
    }
  }
}
```

This will override the default *build-script* job, instructing the generator to generate JS files for the "boot" and the additional "settings" part (a single part may be made up of multiple JS files, depending on cross class dependencies with other parts). In the *boot* part, you are repeating the main *:ref:'include <pages/tool/generator_config_ref#include>'* list of class patterns for you application (the example mirrors this list of a standard GUI app). In the *settings* part, you carve out some top-level classes or name spaces that constitute the part you want to specify. In the example, this is just the name of the top-level dialog class.

Add Part Loading to your Class Code

At a suitable spot in your application code, you have to load the *settings* part, e.g. when the "Open Settings Dialog" button is pressed. We put the loading action in the click event listener of the button:

```
var settingsButton = new qx.ui.toolbar.Button("Open Settings Dialog");

settingsButton.addListener("execute", function(e)
{
  qx.io.PartLoader.require(["settings"], function()
  {
    // if the window is not created
    if (!this.__settingsWindow)
    {
      // create it
      this.__settingsWindow = new custom.Settings();
      this.getRoot().add(this.__settingsWindow);
    }

    // open the window
    this.__settingsWindow.center();
    this.__settingsWindow.open();
  }, this);

}, this);
```

The main thing to note here is that upon pressing the "Open Settings Dialog" button *qx.io.PartLoader.require* is invoked to make sure the *settings* part will be loaded (It doesn't hurt to invoke this method multiple times, as the PartLoader knows which parts have been loaded already).

The first argument to the *require* method is a list containing the parts you want to be loaded (just *"settings"* in our example). The second argument specifies the task that should be done once the part is successfully loaded. As you can see, the *custom.Settings* class, which is loaded with this part, is being instantiated.

These are the essential ingredients to set up and use parts in your application. For a general overview of parts in qooxdoo, see this *article*. For full details on the *packages* configuration key, see the *configuration reference*. For a complete application that uses parts, check the Feedreader sources.

Advanced Usage: Part Collapsing

This section reflects part collapsing as it is realized in qooxdoo version 0.8.3 and above.

Motivation and Background

You as the application developer define *parts* to partition your application. qooxdoo's build system then partitions each part into *packages*, so that each part is made up of some of the set of all packages. Each package contains class code, and maybe some more information that pertains to it. So the classes making up a part are spread over a set of packages. Several parts can share one or more packages. This way you obtain maximum flexibility for loading parts in your application code. Whenever a part is requested through the *PartLoader* it checks which packages have already been loaded with earlier parts, and loads the remaining to make the part complete. No class is loaded twice, and no unnecessary classes are loaded with each part.

But there are situations where you might want to give up on this optimal distribution of classes across packages:

- when packages become **too small**; sometimes packages derived with the basic procedure turn out to be too small, and the benefit of loading no unnecessary classes is outweight by the fact that you have to make an additional net request to retrieve them.
- when you know the **order** in which parts are loaded during run time in advance; then it makes sense to be "greedy" in retrieving as many classes as possible in a single package, as other parts needing the same classes

of the (now bigger) package, but are known to load later, can rely on those classes being loaded already, without being affected by the extra classes that get loaded.

These are situations where *part collapsing* is usefull, where packages are merged into one another. This is discussed in the next sections.

How Packages are Merged

(This is a more theoretical section, but it is kept here for the time being; if you are only looking for how-to information, you can skip this section).

During what we call part collapsing, some packages are merged into others. That means the classes that are contained a source package are added to a target package, and the source package is deleted from all parts referencing it.

Obviously, it is crucial that the target package is referenced in all those parts where the source package was referenced originally, so that a part is not loosing the classes of the source package. This is taken care of by the selection process that for any given source package picks an appropriate target package. (Target packages are searched for in the set of already defined packages, and there are no new packages being constructed during the collapsing process).

After the source package has been merged into the target package, and has been removed from all parts, there are two cases:

- For parts that referenced both (source and target) package initially, there is no difference. The same set of classes is delivered, with the only difference that they come in one, as opposed to two, packages.
- Parts that only reference the target package now reference more classes then they really need. But this should be acceptable, as either negligible (in the case of merging packages by size), since the additional weight is marginal; or as without negative effect (in the case of merging by load order), since the "overladen" package is supposed to be loaded earlier with some other part, and will already be available when this part is loaded.

Collapsing By Package Size

Collapsing by package size is straight forward. You can specify a minimal package size (in KB) that applies to all packages of your application. If a package's size, and it is its *compiled* size that matteres here, is beneath this threshold the package will be merged into another. This avoids the problem of too much fragmentation of classes over packages, and trades optimally distributing the classes (to always load only necessary classes) for minimizing net requests (when loading packages for a part).

Collapsing by size is disabled by default. You enable it by specifying size attributes in your parts configuration:

```
"packages" :
{
  "sizes"    :
  {
    "min-package" : 20,
    "min-package-unshared" : 10
  },
  ...
}
```

The *min-package* setting defines a general lower bound for package sizes, the *min-package-unshared*, which defaults to *min-package* if not given, allows you to refine this value specifically for those packages which pertain to only one part.

Collapsing By Load Order

Collapsing by load order is always useful when you know in advance the order of at least some of your parts, as they are loaded during the app's run time. This is e.g. the case when you have a part that uses other parts to do its work (a big dialogue that has sub-controls like a tabview). The enclosing part is always loaded before its sub-parts can be used. Or there is a part that is only accessible after it has been enabled in another part. These situations can be captured by assigning a load order to (some of) your parts in your configuration.

```
"packages" :
{
  "parts"  :
  {
    "boot" :
    {
      "include"   : [ "${QXTHEME}", "app.Application" ]
    },
    "some-part" :
    {
      "include"   : [ "app.Class1", "app.Class2" ],
      "expected-load-order" : 1
    },
    "other-part" :
    {
      "include"   : [ "app.Class3", "app.Class4" ],
      "expected-load-order" : 2
    },
    ...
  },
  ...
}
```

The *boot* part has always the load index 0, as it is always loaded first. The other parts that have a load index (1 and 2 in the example) will be collapsed with the expectation that they are loaded in this order. Parts that don't have an *expected-load-order* setting are not optimized by part collapsing, and there are no assumptions made as to when they are loaded during run time.

The important thing to note here is that the load order you define is **not destructive**. That means that parts are still self-contained and will continue to function *even if the expected load order is changed during run time*. In such cases, you only pay a penalty that classes are loaded with a part that are actually not used by it. But the overall functionality of your application is not negatively affected.

7.4.3 Further Resources

- *Generator Configuration*
- qooxdoo API

7.5 Miscellaneous

7.5.1 Working with Variants

Variants enable the selection and removal of code from the build version. A variant consists of a collection of states from which exactly one is active at load time of the framework. The global map `qxvariants` can be used to select a variant before the Framework is loaded.

Depending on the selected variant a specific code path can be choosen using the `select` method.

The generator is able to set a variant and remove all code paths which are not selected by the variant.

Variants are used to implement browser optimized builds and to remove debugging code from the build version. It is very similar to conditional compilation in C/C++.

Browser optimized builds

qooxdoo tries to hide browser incompatibilities from the application developer. To provide browser independent functionality it is often necessary to use different code on different browsers. Low level code like the key handler have often their own implementation for each supported browser.

The generator selects for browser optimized builds only the code which is needed for one specific browser and removes the unused code. For each supported browser engine an optimized build is generated and on load time the appropriate build is loaded. As a fall back there is always the unoptimized build.

Code like this was very common in older versions of qooxdoo:

```
if (qx.core.Client.getInstance().isMshtml()) {
  // some Internet Explorer specific code
} else if(qx.core.Client.getInstance().isOpera()){
  // Opera specific code
} else {
  // common code for all other browsers
}
```

Using `Variants` the same code looks like this:

```
if (qx.core.Variant.isSet("qx.client", "mshtml")) {
  // some Internet Explorer specific code
} else if(qx.core.Variant.isSet("qx.client", "opera")){
  // Opera specific code
} else {
  // common code for all other browsers
}
```

The variant `qx.client` is always set to the current browser, so this code works exactly like the first version. What is new is that the `generator` knows about variants and is able to optimize the build for one value of a variant and remove the unused code for all other values of the variant.

Config changes

The browser-specific code above let's you distinguish the different browsers inside your application code. In order to serve different versions of your application for specific browsers you have to slightly change your `config.json` to let the generator do the magic.

```
/* part of your "config.json" */
"jobs" :
{
  /* shadow the original "build-script" job and add the needed infos */
  "build-script" :
  {
    /* adding the variants */
    "variants" :
    {
      "qx.client" : [ "gecko", "mshtml", "webkit", "opera" ]
```

```
    },

    "compile-options" :
    {
      "paths" :
      {
        /* overwrite "file" entry to get client-specific file names */
        "file" : "${BUILD_PATH}/script/${APPLICATION}-{qx.client}.js"
      }
    }
  }
}
```

The generator will generate as many versions of your application as the number of values you give in the list value of *qx.client* (4 in this example). To take advantage of these different variations of your app, you use the *{qx.client}* compile macro in the name of the output file, which will be replaced during compilation by the value currently in effect. This way you'll get output files like *myapp-gecko.js*, *myapp-mshtml.js*, ... asf.

If you specify more than one variant with multiple values, e.g.

```
/* multiple variants with multiple values */
"variants" :
{
  "qx.client"   : [ "gecko", "mshtml", "webkit", "opera" ],
  "qx.debug" : ["on", "off"]
}
```

a compile output is produced **for each possible combination** of all the multi-valued variants, e.g. in this case for *{qx.client: gecko, qx.debug:on}, {qx.client:gecko, qx.debug:off}, {qx.client:mshtml, qx.debug:on}, {qx.client:mshtml, ...}*, ... asf.

You would then also use multiple compile macros in the output file name, e.g. *${APPLICATION}-{qx.client}-{qx.debug}.js*, in order to distinguish those different outputs (otherwise one compile output is copied over the other, and you are left with only the output for the last variation).

Removal of debugging code

Often one wants to add additional checks and assertions to the code but don't want the build to suffer from these checks. This can be solved elegantly by using variants too. The variant `qx.debug` with the allowed values `on` and `off` can be used to add debugging code which is only active in the source version and removed from the build version.

Example:

```
function foo(a, b) {
  if (qx.core.Variant.isSet("qx.debug", "on")) {
    if ( (arguments.length != 2) || (typeof a != "string") ) {
      throw new Error("Bad arguments!");
  }
}
```

This check is now only enabled in the source version. By default `qx.debug` is set to `off` in build versions, and `on` in source versions.

Using variants

Variants are used to select certain code paths. Each variant has a name and exactly one value from a limited list of allowed values. The variant names have a namespace prefix to avoid name conflicts. The value of a variant is immutable and once set cannot be altered in the JavaScript code.

Variants can be used in two ways. They can be used to select code using `if` statements or to select whole functions.

Method: select()

If the whole definition of a function should be selected the `select` method can be used as follows:

```
var f = qx.core.Variant.select("qx.client", {
  "gecko": function() { ... },
  "mshtml|opera": function() { ... },
  "default": function() { ... }
});
```

Depending on the value of the

```
"qx.client"
```

variant the corresponding function is selected. The first case is selected if the variant is "gecko", the second is selected if the variant is "mshtml" or "opera" and the third function is the default one. It is selected if none of the other keys match the variant.

Method: isSet()

This method is used to check whether a variant is set to a given value. The first parameter is the name of the variant and the second parameter is the value to check for. Several values can be "or"-combined by separating them with the "|" character. A value of "mshtml|opera" would for example check whether the variant is set to "mshtml" or "opera".

To enable the generator to optimize this selection, both parameters must be string literals.

This method is meant to be used in if statements to select code paths. If the condition of an if statement is only this method, the generator is able to optimize the if statement.

Example:

```
if (qx.core.Variant.isSet("qx.client", "mshtml")) {
  // some Internet Explorer specific code
} else if(qx.core.Variant.isSet("qx.client", "opera")){
  // Opera specific code
} else {
  // common code for all other browsers
}
```

Framework variants

The following variants are being provided by the framework:

Variant	Allowed values	Default value
`qx.client` Client detection	`gecko`, `mshtml`, `opera`, `webkit`	*auto-detected*
`qx.debug` Debugging code	`on`, `off`	`on`
`qx.aspects` Aspect-oriented programming (AOP)	`on`, `off`	`off`
`qx.dynlocale` Dynamic locale switch	`on`, `off`	`on`

Custom variants

You can easily create your own variants by using `qx.core.Variant.define()`

7.5.2 Internationalization

This page describes how to translate either a new or an existing qooxdoo-based application. It shows how to *prepare* the application, *extract* and *translate* the messages and finally *update* and *run* the translated application.

Prepare the Application

To translate an application, all translatable strings must be marked using one of the following functions:

- `this.tr()`: translate a message
- `this.trn()`: translate a message that supports a plural form
- `this.trc()`: translate a message and providing a comment
- `this.marktr()`: mark a string for translation, but do not perform any translation

You can use these methods right away for your own classes if they are derived from `qx.ui.core.Widget` or `qx.application.AbstractGui`. If that's not the case you have to include the mixin qx.locale.MTranslation manually:

```
qx.Class.define("custom.MyClass",
{
  extend : qx.core.Object,
  include : [qx.locale.MTranslation],
  ...
});
```

Note: You can also use `self` instead of `this` when you use the translation features inside a closure e.g. `self.tr()`. See using self for closures for details using `self` as a local variable name.

Example

Change original code like this:

```
var button = new qx.ui.form.Button("Hello World");
```

to:

```
var button = new qx.ui.form.Button(this.tr("Hello World"));
```

In the following, the four methods are explained in more detail:

tr

```
var button = new qx.ui.form.Button(this.tr("Hello World"));
```

Marks the string `Hello World` for translation and returns an instance of `qx.locale.LocalizedString`. The `toString()` method of the returned object performs the actual translation based on the current locale.

There is one *exception* to the simple rule that all strings can just be replaced by wrapping them in an appropriate `this.tr()` function call: if init values of *dynamic properties* are meant to be localizable, the init value has either to be set in the class constructor using `this.tr()`, or `qx.locale.Manager.tr()` has to be used inside the property declaration. See documentation on *Defining an init value* for details.

trn

```
var n = 2;
var label = new qx.ui.basic.Label(this.trn("Copied one file.", "Copied %1 files.", n, n));
```

Translate a message but take differences between singular and plural forms into account. The first argument represents the singular form, while the second argument represents the plural form. If the third argument is 1 the singular form is chosen, if it is bigger than 1 the plural form is chosen. All remaining parameters are the inputs for the format string.

trc

```
var n = 2;
var label = new qx.ui.basic.Label(this.trc("Helpful comment for the translator", "Hello World"));
```

Translate the message as the `tr` method, but providing an additional comment which can be used to add some contextual information for the translator. This meanigful comment hopefully helps the translator at its work to find the correct translation for the given string.

marktr Sometimes it is necessary to mark a string for translation but not yet perform the translation.

```
var s = this.marktr("Hello");
```

Marks the string `Hello` for translation and returns the string unmodified.

Format Strings Since sentences in different languages can have different structures, it is always better to prefer a format string over string concatenation to compose messages. This is why the methods above all support format strings like `Copied %1 files` as messages and a variable number of additional arguments. The additional arguments are converted to strings and inserted into the original message. `%` is used as an escape character and the number following `%` references the corresponding additional argument.

Extract the Messages

After the source code has been prepared, the desired languages of the application may be specified in `config.json`, in the `LOCALES` macro within the global `let` section, for example

```
"let" :
  {
    // ...
    "LOCALES"        : ["de", "fr"]
  },
```

This would add a German and a French translation to the project. For a more exhaustive list of available locales see here.

A run of

```
generate.py translation
```

will generate a `.po` file for each configured locale, with all translatable strings of the application (These files are usually stored in the `source/translation` folder of the application).

If a specified translation does not yet exist, a new translation file will be created. In this example two files `source/translation/de.po` and `source/translation/fr.po` would be created.

If such a file already exists, the newly extracted strings will be merged with this file, retaining all existing translations.

Therefore, you can re-run `generate.py translation` as often as you want. You should re-run it at least whenever you introduced new translatable strings into the source code, so they will be added to the .po files (s. further *down*).

Translate the Messages

These `.po` files are the actual files you - or your translator ;-) - would have to edit. Since qooxdoo internally uses well-established tools and formats for internationalization ("gettext" via polib), any "po"-aware editor or even a simple text editor can be used.

Some of the programs that support manipulation of `.po` files are:

- Poedit (Windows, Mac OS X, Linux)
- LocFactory Editor (Mac OS X)
- Lokalize (Linux)

Update the Application

After editing and saving the `.po` files, the next `generate.py source` run integrates the translations into your application's source version. To get the effect of the new translations it can simply be reloaded in your browser.

If the source code changes, e.g. by adding, removing or changing translatable strings, it can be merged with the existing translation files just by calling `generate.py translation` again. Moreover, each `generate.py source` - or `generate.py build` if you are about to deploy your application - will pick up all current translatable strings from the source files and will merge them on the fly with the information from the .po files, using the result for the corresponding build job. This way, the generated application always contains all current translatable strings (But of course only those from the .po files can have actual translations with them).

Run the translated Application

By default the application tries to use the browser's default language. You can change the language of the application by using `qx.locale.Manager`. For example, the following sets the language of the application to French:

```
qx.locale.Manager.getInstance().setLocale("fr");
```

The qooxdoo widgets are supposed to update their contents on a locale change. Custom widgets may have to be modified to allow for an update on locale change. To inform the application of a language change, qooxdoo fires a `changeLocale` event.

A widget that needs custom update logic may listen to this event:

```
qx.locale.Manager.getInstance().addListener("changeLocale", this._update, this);
```

7.5.3 Image clipping and combining

qooxdoo integrates the support for clipping and combining images in the framework and both features are heavily used within the framework mainly in the different themes like *appearance or decoration theme*.

Setup

Note: To be able to use image clipping and combining you need an installed *ImageMagick* package.

To use the two features you have to create a config file which can be used by the generator to clip or combine images. Altough it is possible to integrate the jobs for clipping and combining in your `config.json` file of your application, **the better way** is to create an own config file for the image manipuations to separate it from the application configuration.

Note: It is recommended to use the same file name for the config file as in the core framework to better reflect its purpose: `image.json`

At the first look the configuration file for the image jobs is basically the same as a normal application configuration file.

```
{
  "jobs" :
  {
    "common" :
    {
      "let" :
      {
         "RESPATH" : "./source/resource/APPLICATION_NAME"
      },

      "cache" :
      {
         "compile" : "../../cache"
      }
  }
}
```

The described `common` is used to setup the basic settings which are used by the specific jobs `image-clipping` and `image-combine` which are described at the following sections.

Image clipping

Clipping images is needed whenever you have a base image, e.g. a complete image for your button with rounded borders, to strip them into several parts.

Note: Mainly, the clipping is needed to prepare the source image for the use as a `baseImage` for the `grid` decorator. All clipped images of the core framework are used as baseImages for grid decorators.

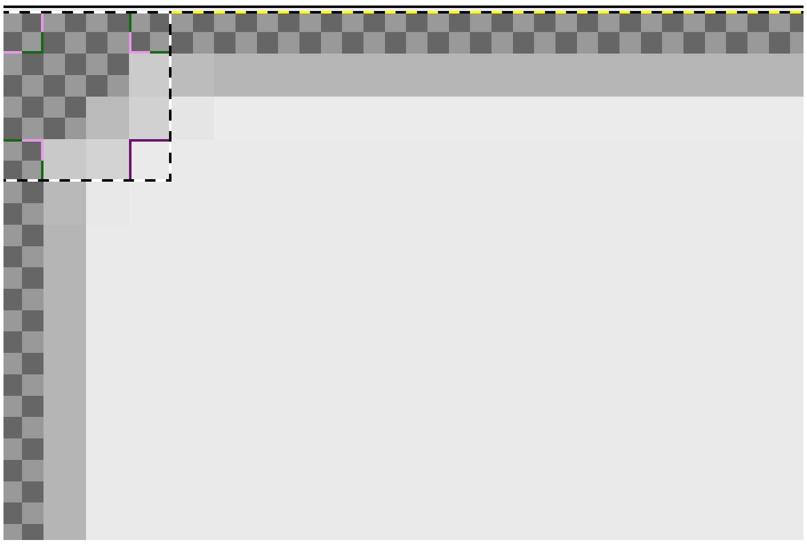
```
"image-clipping" :
{
   "extend" : ["common"],

   "slice-images" :
   {
      "images" :
      {
         "${RESPATH}/image/source/groupBox.png" :
         {
            "prefix" : "../../clipped/groupBox",
            "border-width" : 4
         }
      }
```

```
    }
}
```

Each entry in the `images` block represents one source image to clip.

- value of the key has to be the path to this image
- the `prefix` entry will set the filename for all of your splitted images. The resulting images will follow the rule `prefix+imagepart` where `imagepart` will be e.g. `tl` or `br` (for top-left and bottom-right)
- the entry `border-width` is to define the part of the image which the rounded border occupies. If you look at your baseImage you can determine the "border-width" by select a rectangle (which your graphic tool) which occupies the rounded border completely

For the case `border-width`: One image says more than thousand words :)



The selection rectangle has the size of 4 x 4 pixels, thus the `border-width` value of 4. Differing border width values for each of the four sides are also supported. In that case, the value for border-width must be an array containing the four values in this order: top, right, bottom, left.

Note: For more information see the *slice-image* section.

Image combining

Opposite to image clipping the image combining takes multiple images as source and generates one `combined` image out of them.

```
"image-combine" :
{
   "extend" : ["common"],

   "combine-images" :
   {
      "images" :
      {
         "${RESPATH}/image-combined/combined.png":
         {
            "prefix" : [ "${RESPATH}" ],
            "layout" : "vertical",
```

```
            "input"  :
            [
              {
                "prefix" : [ "${RESPATH}" ],
                "files" : [ "${RESPATH}/image/clipped/groupBox*.png" ]
              }
            ]
         }
      }
   }
}
```

Basically the structure is the same as for the `image-clipping` jobs. Let's take a look at the details.

- value of the key has to the path of the combined image to create
- `files` is an array which takes the several images to combine as arguments - the use of wildcards like `*` or `[tb]` are allowed
- the `layout` key takes the two possible values `horizontal` or `vertical` and determines the alignment of the source images inside the combined images

Note: The layout depends on the sizes of the source images. Best suited for combining are always images with the same sizes. For most cases the `horizontal` layout is the better choice

Note: For more information take a look at the *combine-images* section.

Run image jobs

If you are finished with the definition of your images to clip and/or to combine you can use the `generator` to actually let them created for you.

```
./generate.py -c image.json image-clipping
```

```
./generate.py -c image.json image-combine
```

If you include the following job in your `image.json` jobs list

```
"images" :
{
   "run" : [ "image-clipping", "image-combine" ]
},
```

the execution of

```
./generate.py -c image.json images
```

will run both jobs at once.

Benefits

There are several benefits for setting the image clipping and combining up

- less HTTP requests meaning better performance when using combined images
- widgets using the `grid` decorator are easier to use. If you do not use clipping you have to slice the baseImage and name the parts manually

- state changes are faster with combined images as the browser does not have to change the source if the displayed image. Instead he only changes the value of the CSS property `background-position` to display the desired part of the combined image

7.5.4 Writing API Documentation

For documenting the qooxdoo API special comments in the source code (so-called "doc comments") are used. The doc comments in qooxdoo are similar to JSDoc comments or Javadoc comments. To account for some qooxdoo specific needs there are certain differences to the two systems mentioned above.

The structure of a documentation comment

A doc comment appears right before the structure (class, property, method or constant) it describes. It begins with `/**` and ends with `*/`. The rows in between start with a `*` followed by the text of the particular row. Within this frame there is a description text at the beginning. Afterwards attributes may follow, describing more aspects.

Description texts may also include HTML tags for a better structuring.

An example:

```
/**
 * Shows a message to the user.
 *
 * @param text {string} the message to show.
 */
showMessage : function(text) {
  ...
}
```

This comment describes the method `showMessage`. At the beginning there is a short text, describing the method itself. A `@param` attribute follows, describing the parameter `text`.

The docgenerator recognizes the following structures:

```
/** Class definitions (resp. constructors). */
qx.Class.define("mypackage.MyClass",
{
  extend : blubb.MySuperClass,

  construct : function() {
    ...
  }
});

/** Property definitions. */
properties :
{
   "myProperty" :
    {
       check : "Number",
       init : 0
    }
},

/** Method definitions. */
members :
{
```

```
  myMethod : function(bla, blubb)
  {
    ...
  }
},

/** Static method definitions. */
statics :
{
  myStaticMethod : function(bla, blubb)
  {
    ...
  },

  MY_CONSTANT : 100
},
```

The class description is taken as the first comment in the file which starts with `/**`. Therefore if you have a comment at the start of the file which has a first line of `/**********`, that will be taken as the class description, overriding any comment above the class itself. Therefore use `/* *********` or `/* ==========` etc.

Inline Markup

Running text can be formatted using inline markup which uses special characters around the target text:

- *strong* (will render as **strong**)
- __emphasis__ (will render as *emphasis*)

There is no escape character, so in order to e.g. enter a literal "@" into the text, use the HTML entity equivalent ("*@*" in this case).

Supported attributes

Within a doc comment the following attributes are supported:

@param (only for methods and constructors):

Describes a parameter. `@param` is followed by the name of the parameter. Following that is the type in curly brackets (Example: `{Integer}`), followed by the description text. Types are described more in detail in the next section.

When the parameter is optional, the curly brackets include the default value in addition to the type. The default value implies the value that has to be passed in, in order to get the same effect as when omitting the parameter. Example: `{Boolean ? true}`

You can also define multiple possible types. Example: `{Boolean | Integer ? 0}`

@return (only for methods):

Describes the return value. After the `@return` comes the type in curly brackets followed by the description text.

@throws (only for methods and constructors):

Describes in which cases an exception is thrown.

@see:

Adds a cross reference to another structure (class, property, method or constant). The text is structured as follows: At first comes the full name of the class to link to. If you want to link to a property, method or constant, then a `#` comes, followed by the name of the property, method or constant.

If you refer to a structure within the same class, then the class name may be omitted. If you refer to a class in the same package, then the package name before the class may be omitted. In all other cases you have to specify the fully qualified class name (e.g. `qx.ui.table.Table`).

Some examples:

- `qx.ui.form.Button` refers to the class `Button` in the package `qx.ui.form`.
- `qx.constant.Type#NUMBER` links to the constant `NUMBER` of the class `qx.constant.Type`.
- `qx.core.Init#defineMain` refers to the method `defineMain` in the class `qx.core.Init`

After this target description an alternative text may follow. If missing the target description is shown.

@link:

The `@link` attribute is similar to the `@see` attribute, but it is used for linking to other structures within description texts. Unlike the other attributes, the `@link` attribute is not standalone, but in curly brackets and within the main description text or a description text of another attribute.

@signature:

sometimes the API documentation generator is not able to extract the method signature from the source code. This is for example the case, when the method is defined using `variants` of if the method is assigned from a method constant like `qx.lang.Function.returnTrue`.

In these cases the method signature can be declared inside the documentation comment using the @signature attribute.

Example:

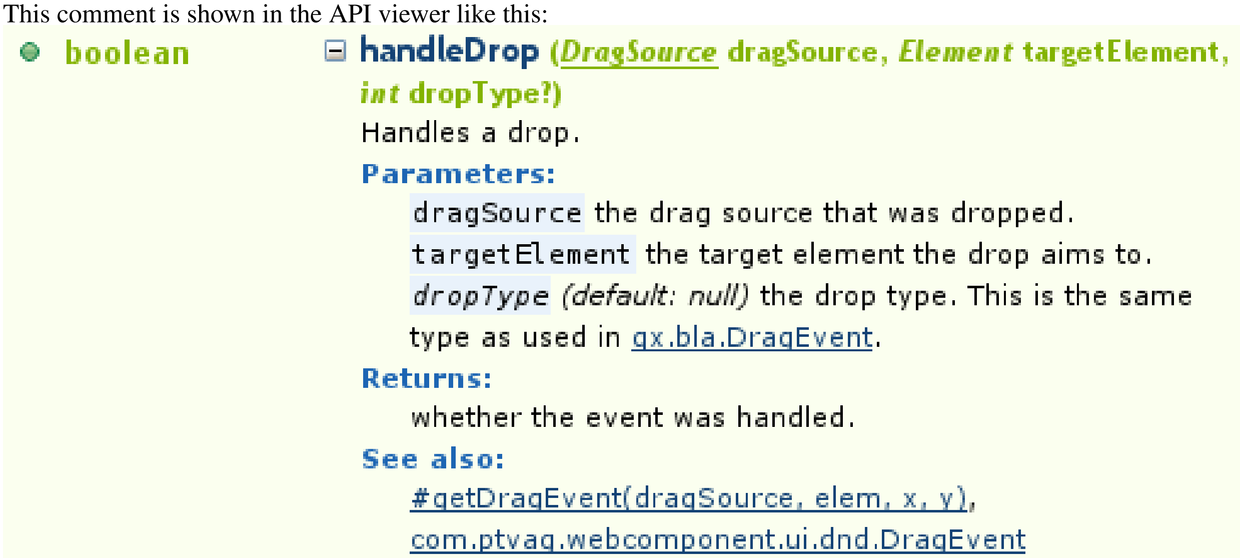
```
members :
  {
    /**
     * Always returns true
     *
     * @return {Boolean} returns true
     * @signature function()
     */
    sayTrue: qx.lang.Function.returnTrue;
  }
```

Example

Example for a fully extended doc comment:

```
/**
* Handles a drop.
*
* @param dragSource {qx.bla.DragSource} the drag source that was dropped.
* @param targetElement {Element} the target element the drop aims to.
* @param dropType {Integer ? null} the drop type. This is the same type as used in
*        {@link qx.bla.DragEvent}.
* @return {Boolean} whether the event was handled.
* @throws if the targetElement is no child of this drop target.
*
* @see #getDragEvent(dragSource, elem, x, y)
* @see com.ptvag.webcomponent.ui.dnd.DragEvent
*/
handleDrop : function(dragSource, targetElement, dropType) {
  ...
};
```

This comment is shown in the API viewer like this:



Handling of data types

Because JavaScript has no strong typing, the types of the parameters accepted by a method may not be read from the method's definition. For showing the accepted types in the API documentation the data type may be specified in the doc attributes `@param` and `@return`.

The following types are accepted:

- Primitive: `var`, "void", "undefined"
- Builtin classes: `Object`, `Boolean`, `String`, `Number`, `Integer`, `Float`, `Double`, `Regexp`, `Function`, `Error`, `Map`, `Date` and `Element`
- Other classes: Here the full qualified name is specified (e.g. `qx.ui.core.Widget`). If the referenced class is in the same package as the currently documented class, the plain class name is sufficient (e.g. `Widget`).

Arrays are specified by appending one or more `[]` to the type. E.g.: `String[]` or `Integer[][]`.

__init__.js Files

While using doc comments in class files where they are interleaved with the class code is straight forward, this is not so trivial if you want to provide documentation for a *package*, i.e. a collection of classes under a common name space (like *qx.ui.core*, *qx.util*, etc.).

In order to fill this gap you can add a __init.js__ file to a package. This file should only contain a single doc comment that describes the package as a whole. These files are then scanned during a `generate.py api` run and the documentation is inserted at the package nodes of the resulting documentation tree.

7.5.5 Reporting Bugs

Note: Please see the general document on How to report bugs

7.5.6 An Aspect Template Class

Here is a code template which you may copy to your application namespace and adapt it to implement aspects in your qooxdoo application. For a far more advanced sample look at qx.dev.Profile.

```
/**
 * AspectTemplate.js -- template class to use qooxdoo aspects
 *
 * This is a minimal class template to show how to deploy aspects in qooxdoo
 * applications. For more information on the aspect infrastructure see the API
 * documentation for qx.core.Aspect.
 *
 * You should copy this template to your application namespace and adapt it to
 * your needs. See the comments in the code for further hints.
 *
 * To enable the use of your aspect class, some extra settings need to be done
 * in your configuration file.
 *   * Add a require of your aspects class to qx.Class
 *   * Set the variant qx.aspects to on
 *   * Set the setting qx.enableAspect to true
 *
 */

/* ************************************************************************

#require(qx.core.Aspect)
#ignore(auto-require)

************************************************************************ */

/** Adapt the name of the class */
qx.Bootstrap.define("your.namespace.YourAspectClass", {

  /** The class definition may only contain a 'statics' and a 'defer' member */
  statics :
  {

    __counter : 0,  // Static vars are possible

    /**
     * This function will be called before each function call.
     *
```

```
     * @param fullname {String} Full name of the function including the class name.
     * @param fcn {Function} Wrapped function.
     * @param type {String} The type of the wrapped function (static, member, ...)
     * @param args {Arguments} The arguments passed to the wrapped function.
     */
    atEnter: function(fullName, fcn, type, args)
    {
      console.log("Entering "+fullName);  // Adapt this to your needs
    },

    /**
     * This function will be called after each function call.
     *
     * @param fullname {String} Full name of the function including the class name.
     * @param fcn {Function} Wrapped function.
     * @param type {String} The type of the wrapped function (static, member, ...)
     * @param args {Arguments} The arguments passed to the wrapped function.
     * @param returnValue {var} return value of the wrapped function.
     */
    atExit: function(fullName, fcn, type, args, returnValue)
    {
      console.log("Leaving "+fullName);  // Adapt this to your needs
    }

  },

  defer : function(statics)
  {
    /**
     * Registering your static functions with the aspect registry. For more
     * information see the API documentation for qx.core.Aspect.
     *
     * @param fcn {Function} Function from this class to be called.
     * @param position {String} Where to inject the aspect ("before" or "after").
     * @param type {String} Which types to wrap ("member", "static", "constructor",
     *                       "destructor", "property" or "*").
     * @param name {RegExp} Name(pattern) of the functions to wrap.
     */
    qx.core.Aspect.addAdvice(statics.atEnter, "before", "*", "your.namespace.*");
    qx.core.Aspect.addAdvice(statics.atExit, "after",  "*", "your.namespace.*");
  }

});
```

A job in your configuration could look something like this:

```
"source" :
      {
        "require" :
        {
          "qx.Class" : ["aspects.Aop"]
        },

        "variants" :
        {
          "qx.aspects" : [ "on" ]
        },

        "settings" :
```

```
    {
      "qx.enableAspect" : true
    }
  }
```

If you need some more information on configuring the generator, take a look at the *Reference Listing of Config Keys*.

7.5.7 Internet Explorer specific settings

This page describes all settings which are used/implemented by qooxdoo to workaround/solve IE-specific issues.

Document Mode in IE8

qooxdoo uses *Internet Explorer 8 standard mode* as the default for all generated applications. This is achieved by setting a XHTML 1.2 Doctype to all *index.html* files provided by the framework.

Using alpha-transparent PNGs

IE7 and IE8 have built-in support for loading alpha-transparent PNGs. qooxdoo however does use the AlphaImageLoader for all IE versions whenever a PNG image has to be loaded. This has several reasons:

- Performance issues in IE8 - native alpha PNG support is slower when running IE8 standards mode
- Rendering bug in IE - reported at Bug #1287

URL-Rewriting under HTTPS

Every IE version does show a *Mixed Content* warning whenever a resource (image, script, etc.) is loaded with a regular HTTP request when the containing page runs under HTTPS. However, this useful warning is also appearing in situations it is not acceptable, e.g. for relative paths like *./path/to/my/resource*. In order to solve these issues every URL of a resource managed by qooxdoo is rewritten in IE under HTTPS to an absolute URL to prevent the warning. See Bug #2427 for more details.

CHAPTER

EIGHT

TOOLING

8.1 Generator Introduction

8.1.1 Generator Overview

This is a high-level overview of some important generator features.

Quick links:

- *Generator Usage*
- *Configuration file details*

Configuration

- Load project configuration from JSON data file
- Each configuration can define multiple so named jobs
- Each job defines one action with all configuration
- A job can extend any other job and finetune the configuration
- Each execution of the generator can execute multiple of these jobs

Cache Support

- Advanced multi-level caching system which dramatically reduces the runtime in repeated calls.
- The cache stores all data on the disk using the `cpickle` Python module.
- Invalidation of cache files happens through a comparision of modification dates.
- Cache filenames are generated through SHA1 (hex) to keep them short and unique.
- There is memory-only caching as well. It is used for dependencies and meta data.
- The system supports caching for:
  - extracted meta data
  - syntax tokens
  - syntax trees
  - class dependencies

   - compiler results
   - api data
   - localizable strings

Class Selection

- Use include/exclude lists to define the classes to use.
- Has support for simple expressions inside each include or exclude definition e.g. `qx.*`.
- The smart mode (default) includes/excludes the defined classes and their dependencies. This mode also excludes all classes only required by the excluded classes.
- The other mode is toggled using a = prefix. This switches to a mode where exactly the classes mentioned are included/excluded.
- As a fallback all known classes will be added when no includes are defined.

Variants

- It is possible to generate multiple variant combinations. This means that a single job execution can create multiple files at once using different so-named variant sets. Variants are combinable and all possible combinations are automatically created. For example: gecko+debug, mshtml+debug, gecko+nodebug, mshtml+nodebug
- The system supports placeholders in the filename to create filenames based on variant selection [TBD].

API Data

- Creation of split API data which loads incrementally as needed.
- Creation of API index containing all relevant names of the API (e.g. classes, properties, functions, events, ...)

Internationalisation

- Creation and update of “po” files based on the classes of any namespace.
- Generation of JavaScript data to be included into application
- Dynamic creation of localization data based on the standardized informations available at unicode.org. The “main” package of CLDR which is used, is locally mirrored in the SDK.

Parts

- Each part can be seen as a logical or visual group of the application.
- Each part may result into multiple packages (script output files).
- The number of packages could be exponential to the number of parts (but through the optimization this is often not the case).
- It is possible to automatically collapse any number parts (e.g. merging the packages used by a part). Such an important part may be the one which contains the initial application class (application layout frame) or the splashscreen. Collapsing reduces the number of packages (script files) for the defined parts. However collapsing badly influences the fine-grained nature of the package system and should be ommitted for non-initial parts normally.

- Further optimization includes support for auto-merging small packages. The relevant size to decide if a package is too small, is the minimum compiled size which is defined by the author of the job. The system calculates the size of each package and tries to merge packages automatically.
- The parts can be used in combination with the include/exclude system. Includes can be used to select the classes to use.
- By default all classes mentioned in the parts are added to the include list. It is possible to override this list.
- All global excludes listed are also respected for the parts.

8.1.2 Generator Usage

The generator is a command-line utility which serves as the single entry point front-end for all qooxdoo tool chain functions (nearly; there are a few functions that are available through other programs, but these really serve special-case purposes).

The generator is started to execute various jobs. Those jobs represent the feature set of the tool chain. This page is about how to invoke the generator.

Files and Folder Structure

The qooxdoo SDK has a dedicated `tool` folder that contains all elements that make up the tool chain. The general structure is like this:

```
tool
   |- app    -- helper apps
   |- bin     -- stand-alone programs and scripts
   |- data   -- various data files
   |- pylib  -- Python modules
```

The generator is actually the program under `tool/bin/generator.py`.

generate.py

To make it easier to invoke the generator, each library in the SDK (framework, applications, components) contains a `generate.py` script that is really just a proxy for the generator itself. It is also part of each project structure created by the *create-application.py* wizard. The aim is to hide the actual path to the generator program.

Command-line Options

Since the generator is nearly complete driven by its config files, there are very few command-line options:

```
shell> generator.py -h
Usage: generator.py [options] job,...

Arguments:
  job,...               a list of jobs (like 'source' or 'copy-files',
                        without the quotes) to run
  x                     use 'x' (or some undefined job name) to get a
                        list of all available jobs from the configuration file

Options:
  -h, --help            show this help message and exit
  -c CFGFILE, --config=CFGFILE
```

```
                        path to configuration file containing job definitions
                        (default: config.json)
  -q, --quiet           quiet output mode (extra quiet)
  -v, --verbose         verbose output mode (extra verbose)
  -w, --config-verbose  verbose output mode of configuration processing
  -l FILENAME, --logfile=FILENAME
                        log file
  -s, --stacktrace      enable stack traces on fatal exceptions
  -m KEY:VAL, --macro=KEY:VAL
                        define/overwrite a global 'let' macro KEY with value
                        VAL
```

The most important options are the path of the config file to use (*-c* option), and the list of jobs to execute. The *-m* option allows Json-type values, scalars like strings and numbers, but also maps *{...}* and lists *[...]*.

Configuration Files

The singe most-important way to control the actions of the generator is through specialized config files. These files have a JSON syntax and contain the definitions for the various jobs the generator is supposed to execute. There is a *whole section* in this manual dedicated to these config files.

Usage Patterns

As a few quick hints at how you would invoke the generator, here are the most common use cases. All these examples name a single job to run, and rely on the availability of the default config file `config.json` in the current directory:

- `generate.py source` – when you just started to create your application and every time you have added new classes to it.
- `generate.py build` – when you have completed your application and/or want to create an optimized, deployable version of it.
- `generate.py api` – when your application is getting complex and/or you want to have a local version of the standard Apiviewer application that includes the documentation of all of your application classes.
- `generate.py test` – when you have created unit test classes for your application and want to run them in the Testrunner frame application.

The *Hello World* tutorial will give the complete steps how to start a project and get going.

Default Jobs

Arguments like `source` or `api`, as shown in the previous section, are so called *jobs* in qooxdoo lingo. If you are working on a skeleton-based application you automatically get a whole list of such pre-defined jobs to work with. For a quick overview, invoke the generator script with an undefined job argument, like

```
generate.py X
```

This gives you a list of all jobs available through your current config file, many of them with a few words of explanation about what they do:

```
- api            -- create api doc for the current library
- build          -- create build version of current application
- clean          -- remove local cache and generated .js files (source/build)
- distclean      -- remove the cache and all generated artefacts of this library (source, build, ...)
- fix            -- normalize whitespace in .js files of the current library (tabs, eol, ...)
```

```
- inspector     -- (since 0.8.2) create an inspector instance in the current library
- lint          -- check the source code of the .js files of the current library
- migration     -- migrate the .js files of the current library to the current qooxdoo version
- pretty        -- pretty-formatting of the source code of the current library
- profiling     -- includer job, to activate profiling
- source        -- create source version of current application
- source-all    -- create source version of current application, with all classes
- test          -- create a test runner app for unit tests of the current library
- test-source           -- create a test runner app for unit tests (source version) of the current lib
- translation           -- create .po files for current library
```

For an exhaustive reference of these default jobs, see the *default jobs page*.

8.2 Generator Configuration

8.2.1 Generator Configuration File

Overview

The configuration file that drives the generator adheres to the JSON specification. It has the following general structure:

```
{
  "jobs" :
  {
    "job1" : { ... },
    "job2" : { ... },
    ...
    "jobN" : { ... }
  }
}
```

The job names `job1`, ..., `jobN` are freely chooseable names but must form a valid key. JavaScript-style comments (/.../ and %//%...) are permissible but only in rather robust places, like after a comma or directly after opening or before closing parentheses, but e.g. not between a key and its value.

Quick links:

- *Reference Listing of Config Keys*
- *Configuration Macro Reference*
- *Configuration Detail Articles*
- `Implementation Background Information`

Example

Here is an example of a minimal config file that defines a single job to create the source version of an application:

```
{
  "jobs" :
  {
    "source" :
    {
      "let" :
      {
```

```
        "QOOXDOO_PATH" : "../..",
        "APPLICATION" : "custom"
      },

      "library" :
      [
        {
          "manifest"     : "${QOOXDOO_PATH}/framework/Manifest.json"
        },
        {
          "manifest"     : "./Manifest.json"
        }
      ],

      "compile-options" :
      {
        "paths" :
          {
            "file" : "./source/script/${APPLICATION}.js"
          }
      },

      "compile" : { "type" : "source" },

      "require" :
      {
        "qx.log.Logger" : ["qx.log.appender.Native"]
      },

      "settings" :
      {
        "qx.application" : "${APPLICATION}.Application"
      },

      "cache" :
      {
        "compile" : "../../cache2"
      }
    }
  }
}
```

Syntax

Apart from the general Json rules, you can place '=' in front of job and key names, to indicate that this feature should prevail as specified when configs get merged. See *here* for more details on that. The config system also allows the use of *macros*, details of which can be found *here*.

Valid Job Keys

The value of each job is a map where the keys are **not** freely chooseable, but are predefined.

Keys can be grouped into several categories:

- `structure-changing` - Keys that influence the configuration itself, e.g. the contents or structure of jobs, the job queue, or the config file as a whole (e.g. *extend, include (top-level), run*).

- `actions` - Keys that if present trigger a certain action in the generator, which usually results in some output (e.g. *compile, api, localize*).
- `input/output-setting` - Keys that specify input (e.g. classes or ranges of classes to deal with) and output (e.g. packaging, variants) options (e.g. *library, require, include*).
- `runtime-settings` - Keys pertaining to the working needs of the generator (e.g. *cache*).
- `miscellaneous` - Keys that don't fall in any of the other categories (e.g. *desc*).

First, here is an overview table, to list all possible keys in a job (unless otherwise noted). Below that you'll find a structured listing of all possible configuration keys in their respective context, with links to further information for each key.

Action Keys	**Description**
api	Triggers the generation of a custom Apiviewer application.
clean-files	Delete files and directories from the file system.
collect-environment-info	Prints various infos about the qooxdoo environment (version etc.)
combine-images	Triggers creation of a combined image file that contains various images.
compile	Triggers the generation of a source or build version of the app.
copy-files	Triggers files/directories to be copied.
copy-resources	Triggers the copying of resources.
fix-files	Fix white space in source files.
lint-check	Check source code with a lint-like utility.
migrate-files	Migrate source code to the current qooxdoo version.
pretty-print	Format source files.
provider	Collects classes, resources and dependency info in a directory tree.
shell	Triggers the execution of one or more external command(s).
simulate	Triggers the execution of a suite of integration tests.
slice-images	Triggers cutting images into regions.
translate	Triggers updating of .po files.
Structure-changing Keys	**Description**
default-job (top-level)	Default job to be run.
export (top-level)	List of jobs to be exported to other config files.
extend	Extend the current job with other jobs.
include (top-level)	Include external config files.
jobs (top-level)	Define jobs.
let	Define macros.
let (top-level)	Define default macros.
run	Define a list of jobs to run.
Input/Output-setting Keys	**Description**
add-script	Includes aritrary URIs to be loaded by the app.
asset-let	Defines macros that will be replaced in #asset hints.
compile-options	Various options that taylor the *compile* action.
dependencies	Fine-tune dependency processing.
exclude	Exclude classes from processing of the job.
include	Include classes to be processed in the job.
library	Define libraries to be taken into account for this job.
packages	Define packages for this app.
require	Define prerequisite classes (load time).
settings	Define qooxdoo settings.
use	Define prerequisite classes (run time).
variants	Define variants for the current app.

Continued on next page

Table 8.1 – continued from previous page

Runtime-setting Keys	Description
cache	Define the path to the cache directory.
log	Tailor log output options.
Miscellaneous Keys	**Description**
desc	A descriptive string for the job.
name (top-level)	A descriptive string for the configuration file.

Listing of Keys in Context

This shows the complete possible contents of the top-level configuration map. Further information is linked from the respective keys.

- *name* A name or descriptive text for the configuration file.
- *include* Include external config files. Takes a list of maps, where each map specifies an external configuration file, and options how to include it. (See special section on the *include key*)
- *let* Define default macros. Takes a map (see the description of the job-level 'let' further down). This let map is included automatically into every job run. There is no explicit reference to it, so be aware of side effects.
- *export* List of jobs to be exported if this config file is included by another.
- *default-job* The name of a job to be run as default, i.e. when invoking the generator without job arguments.
- *jobs* Map of jobs. Each key is the name of a job.
  - *<jobname>* Each job's value is a map describing the job. The describing map can have any number of the following keys:
    - *add-script* A list of URIs that will be loaded first thing when the app starts.
    - *api* Triggers the generation of a custom Apiviewer application.
    - *asset-let* Defines macros that will be replaced in #asset hints in source files. (See special section on the *"asset-let" key*).
    - *cache* Define the path to cache directories, most importantly to the compile cache. (See special section on the *"cache" Key* key).
    - *clean-files* Triggers clean-up of files and directories within a project and the framework, e.g. deletion of generated files, cache contents, etc.
    - *collect-environment-info* Collects various information about the qooxdoo environment (like version, cache, etc.) and prints it to the console.
    - *combine-images* Triggers creation of a combined image file that contains various images.
    - *compile* Triggers the generation of a source or build version of the application.
    - *compile-options* Define various options that influence compile runs (both source and build version).
    - *copy-files* Triggers files/directories to be copied, usually between source and build version.
    - *copy-resources* Triggers the copying of resources, usually between source and build version.
    - *dependencies* Fine-tune the processing of class dependencies.
    - *desc* A string describing the job.
    - *exclude* List classes to be excluded from the job. Takes an array of class specifiers.

* *extend* Extend the current job with other jobs. Takes an array of job names. The information of these jobs are merged into the current job description, so the current job sort of "inherits" their settings. (See the special section on *"extend" semantics*).
* *fix-files* Fix white space in source files.
* *include* List classes to be processed in the job. Takes an array of class specifiers.
* *let* Define macros. Takes a map where each key defines a macro and the value its expansion. (See the special section on *macros*).
* *library* Define libraries to be taken into account for this job. Takes an array of maps, each map specifying one library to consider. The most important part therein is the "manifest" specification. (See special section on *Manifest files*).
* *lint-check* Check source code with a lint-like utility.
* *log* Tailor log output of job.
* *migrate-files* Migrate source code to the current qooxdoo version.
* *packages* Define packages for the application. (See special section on *packages*).
* *pretty-print* Triggers code beautification of source class files (in-place-editing). An empty map value triggers default formatting, but further keys can tailor the output.
* *provider* Collects classes, resources and dependency information and puts them in a specific directory structure under the `provider` root.
* *require* Define prerequisite classes needed at load time. Takes a map, where the keys are class names and the values lists of prerequisite classes.
* *run* Define a list of jobs to run in place of the current job. (See the special section on *"run" semantics*).
* *settings* Define qooxdoo settings for the generated application.
* *shell* Triggers the execution of one or more external command(s).
* *simulate* Triggers the execution of a GUI test (simulated interaction) suite.
* *slice-images* Triggers cutting images into regions.
* *translate* Re-)generate .po files from source classes.
* *use* Define prerequisite classes needed at run time. Takes a map, where the keys are class names and the values lists of prerequisite classes.
* *variants* Define variants for the generated application.

8.2.2 Generator Configuration Articles

This page contains various articles related to the generator JSON configuration.

Path Names

A lot of entries in a config file take path names as their values (top-level "include", "manifest" keys of a library entry, output path of compile keys, asf.). Quite a few of them, like the top-level include paths, are interpreted **relative** to the config file in which they appear, and this relation is retained no matter from where you reference the config file.

This might not hold true in each and every case, though. For some keys you might have to take care of relative paths yourself. The authoritative reference is always the corresponding documentation of the *individual config keys*. If a key takes a path value it will state if and how these values are interpreted. Please check there.

A good help when dealing with paths is also to use macros, if you need to abstract away from a value appearing multiple times. E.g.

```
"let" : {"MyRoot": ".", "BUILD_PATH" : "build"}
"myjob" : { ... "build_dir" : "${MyRoot}/${BUILD_PATH}" ... }
```

This should make it more intuitive to maintain a config file.

Note: Implementor's note:

The configuration handler (`generator/config/Config.py`) handles relative paths in the obvious cases, like for the `manifest` entries in the `library` key, or in the top-level `include` key. But it cannot handle all possible cases, because it doesn't know beforehand which particluar key represents a path, and which doesn't. In a config entry like `"foo" : "bar"` it is hard to tell whether `bar` represents a relative file or directory. Therefore, part of the responsibility for relative paths is offloaded to the action implementations that make use of the particular keys.

Since each config key, particularly action keys, interpret their corresponding config entries, they know which entries represent paths. To handle those paths correctly, the `Config` module provides a utility method `Config.absPath(self, path)` which will calculate the absolute path from the given path relative to the config file's location.

File Globs

Some config keys take file paths as their attributes. Where specified, *file globs* are allowed, as supported by the underlying Python module. File globs are file paths containing simple metacharacters, which are similar to but not quite identical with metacharacters from regular expressions. Here are the legal metacharacters and their meanings:

Metacharacter	Meaning
*	matches any string of zero or more characters (regexp: .*)
?	matches any single character (regexp: .)
[]	matches any of the enclosed characters; character ranges are possible using a hyphen, e.g. [a-z] (regexp: <same>)

Examples

Given a set of files like `file9.js, file10.js, file11.js`, here are some file globs and their resolution:

File Glob	Resolution
file*	file9.js, file10.js and file11.js
file?.js	file9.js
file1[01].js	file10.js and file11.js

Class Data

Besides code a qooxdoo application maintains a certain amount of data that represents some sort of resources. This might be negligible for small to medium size applications, but becomes significant for large apps. The resources fall roughly into two categories,

- **Internationalization (I18N) Data** This comprises two kinds of data:
  - translated strings
  - locale information (also CLDR data, such as currency, date and time formats, asf.)
- **File Resources**

  - static files like PNG and GIF graphics, but also HTML and CSS files, sound and multimedia files, asf.

Many of these resources need an internal representation in the qooxdoo app. E.g. translated strings are stored as key:value pairs of maps, and images are stored with their size and location. All this data requires space that shows up in sizes of application files, as they are transfered from server to browser.

The build system allows you to tailor where those resources are stored, so you can optimize on your network consumption and memory footprint. Here is an overview:

- **source** version: - without dedicated I18N parts:all class data is allocated in the loader - with dedicated I18N parts:class data is in dedicated I18N packages
- **build** version: - without dedicated I18N parts: class data is allocated in each individual package, corresponding to the contained class code that needs it - with dedicated I18N parts: class data is in dedicated I18N packages

The term *"dedicated I18N parts"* refers to the possibility to split translated strings and CLDR data out in separate parts, one for each language (see *packages/i18n-with-boot*). Like with other parts, those parts have to be actively loaded by the application (using qx.io.PartLoader.require).

In the build version without dedicated I18N parts (case 2.1), those class data is stored as is needed by the code of the package. This may mean that the same data is stored in multiple packages, as e.g. two packages might use the same image or translated string. This is to ensure optimal independence of packages among each other so they can be loaded independently, and is resolved at the browser (ie. resource data is stored uniquely in memory).

"cache" Key

Compile cache

The main payload of the *cache* key is to point to the directory for the compile cache. It is very recommendable to have a system-wide compile cache directory so cache contents can be shared among different projects and libraries. Otherwise, the cache has to be rebuilt in each enviornment anew, costing extra time and space.

The default for the cache directory is beneath the system TMP directory. To find out where this actually is either run `generate.py info`, or run a build job with the `-v` command line flag and look for the *cache* key in the expanded job definition, or use this snippet.

The compile cache directory can become very large in terms of contained files, and a count of a couple of thousand files is not unusual. You should take care that your file system is equipped to comply with these demands. Additionally, disk I/O is regularly high on this directory so a fast, local disk is recommendable. Don't use a network drive :-) .

"let" Key

Config files let you define simple macros with the `let` key. The value of a macro can be a string or another JSON-permissible value (map, array, ...). You refer to a macro value in a job definition by using `${<macro_name>}`.

```
"let": {"MyApp" : "demobrowser"}
  ...
  "myjob" : { "settings" : {"qx.application" : "${MyApp}.Application"}}
```

If the value of the macro is a string you can use a reference to it in other strings, and the macro reference will be replaced by its value. You can have multiple macro references in one string. Usually, these macro references will show up in map values or array elements, but can also be used in map keys.

```
"myjob" : {"${MyApp}.resourceUri" : "resource"}
```

If the value of the macro is something other than a string, things are a bit more restrictive. References to those macros can not be used in map keys (for obvious reasons). The reference has still to be in a string, but the macro reference has to be **the only contents** of that string. The entire string will then be replaced by the value of the macro. That means, you can do something like this:

```
"let" : {"MYLIST" : [1,2,3], ...},
  "myjob" : { "joblist" : "${MYLIST}", ...}
```

and the "joblist" key will get the value [1,2,3].

A special situation arises if you are using a **top-level let**, i.e. a *let* section on the highest level in the config file, and not in any job definition. This *let* map will be automatically applied to every job run, without any explicit reference (so be aware of undesired side effects of bindings herein).

When assembling a job to run, the precedence of all the various *let* maps is

```
local job let < config-level let < 'extend' job lets
```

With imported jobs top-level definitions will take precedence over any definitions from the external config file (as if they were the 'first' let section in the chain).

"log" Key

Logging is an important part of any reasonably complex application. The Generator does a fair bit of logging to the console by default, listing the jobs it performs, adding details of important processing steps and reporting on errors and potential inconsistencies. The *log* key lets you specify further options and tailor the Generator console output to your needs. You can e.g. add logging of unused classes in a particular library/name space.

"extend" Key

Job resolution

`extend` and `run` keywords are currently the only keywords that reference other jobs. These references have to be resolved, by looking them up (or "evaluating" the names) in some context. One thing to note here is that job names are evaluated **in the context of the current configuration**. As you will see (see section on *top-level "include"s*), a single configuration might eventually contain jobs from multiple config files, the local job definitions, and zero to many imported job maps (from other config files), which again might contain imported configs. From within any map, only those jobs are referenceable that are **contained** somewhere in this map. Unqualified names (like "myjob") are taken to refer to jobs on the same level as the current job, path-like names (containing "/") are taken to signify a job in some nested name space down from the current level. Particularly, this means you can never reference a job in a map which is "parallel" to the current job map. It's only jobs on the same level or deeper.

This is particularly important for imported configs (imported with a top-level "include" keyword, see further *down*). Those configs get attached to the local "jobs" map under a dedicated key (their "name space" if you will). If in this imported map there is a "run" job (see the *next section*) using unqualified job names, these job names will be resolved using the imported map, not the top-level map. If the nested "run" job uses path-like job names, these jobs will be searched for **relative** to the nested map. You get it?!

Extending jobs

Now, how exactly is a job (let's call this the primary job) treated that says to "extend" another job (let's call this the secondary job). Here is what happens:

- The primary job provides sort of the master definition for the resulting job. All its definitions take precedence.
- The secondary job is searched in the context of the current "jobs" map (see above).

- Keys of the secondary job that are **not** available in the primary job are just added to the job definition.
- Keys of the secondary job that are already present in the primary job and have a scalar value (string, number, boolean) are **discarded**.
- Keys of the secondary job that are already present in the primary job and have a list or map value are **merged**. The extending rules are applied on the element level recursively, i.e. scalar elements are blocked, new elements are added, composed element are merged. That means, those keys accumulate all their inner keys over all jobs in the transitive hull of all extend jobs of the primary job.
- There is a way of **preventing** this kind of merge behaviour: If you prefix a job key with an equal sign (=) no subsequent merging will be done on this key. That means all following jobs that are merged into the current will not be able to alter the value of this key any more.
- Obviously, each secondary job is extended itself **before** being processed in this way, so it brings in its own full definition. As stated before it is important to note that this extending is done in the secondary job's **own** context, which is not necessarily the context of the primary job.
- If there are more than one job in the "extend" list, the process is re-applied **iteratively** with all the remaining jobs in the list. This also means that the list of secondary jobs defines a precedence list: Settings in jobs earlier in the list take precedence over those coming later, so order matters.

Important to note here: **Macro evaluation** takes place only **after** all extending has been done. That is, macros are applied to the fully extended job, making all macro definitions available that have accumulated along the way, with a 'left-to-right' precedence (macro definitions in the primary job take precedence over definitions in secondary jobs, and within the list of secondary jobs, earlier jobs win over subsequent). But in contrast to job names that also means that macros are explicitly **not** evaluated in the original context of the job. This makes it possible to tweak a job definition for a new environment, but can also lead to surprises if you wanted to have some substitution taking place in the original config file, and realize it doesn't.

Job Shadowing and Partial Overriding

Additionally to the above described features, with the configuration system you can

- create jobs in your local configuration with *same names* as those imported from another configuration file. The local job will take precedence and "shadow" the imported job; the imported job gets automatically added to the local job's `extend` list.
- extend one job by another by only *partially specifying* job features. The extending job can specify only the specific parts it wants to re-define. The jobs will then be merged as described above, giving precedence to local definitions of simple data types and combining complex values (list and maps); in the case of maps this is a deep merging process. Here is a sample of overriding an imported job (`build-script`), only specifying a single setting, and relying on the rest to be provided by the imported job of same name:

```
"build-script" : {
  "compile-options" : {
    "code" : {
      "format" : true
    }
  }
}
```

You can again use = to control the merging:

- *selectively block* merging of features by using = in front of the key name, like:

```
...
  {
    "=open-curly" : ...,
```

```
        ...
      }
    ...
```

- override an imported job *entirely* by guarding the local job with = like:

```
"jobs" : {
  "=build-script" : {...},
  ...
}
```

"run" Key

"run" jobs are jobs that bear the `run` keyword. Since these are kind of meta jobs and ment to invoke a sequence of other jobs, they have special semantics. When a `run` keyword is encountered in a job, for each sub-job in the "run" list a new job is generated (so called *synthetic jobs*, since they are not from the textual config files). For each of those new jobs, a job name is auto-generated using the initial job's name as a prefix. As for the contents, the initial job's definition is used as a template for the new job. The `extend` key is set to the name of the current sub-job (it is assumed that the initial job has been expanded before), so the settings of the sub-job will eventually be included, and the "run" key is removed. All other settings from the initial job remain unaffected. This means that all sub-jobs "inherit" the settings of the initial job (This is significant when sub-jobs evaluate the same key, and maybe do so in a different manner).

In the overall queue of jobs to be performed, the initial job is replaced by the list of new jobs just generated. This process is repeated until there are no more "run" jobs in the job queue, and none with unresolved "extend"s.

"asset-let" Key

The `asset-let` key is basically a *macro* definition for `#asset` compiler hints, but with a special semantics. Keys defined in the "asset-let" map will be looked for in *#asset* hints in source files. Like with macros, references have to be in curly braces and prefixed with `$`. So a "asset-let" entry in the config might look like this:

```
"asset-let" :
  {
    "qx.icontheme" : ["Tango", "Oxygen"],
    "mySizes" : ["16", "32"]
  }
```

and a corresponding *#asset* hint might use it as:

```
#asset(qx/icon/${qx.icontheme}/${mySizes}/*)
```

The values of these macros are lists, and each reference will be expanded into all possible values with all possible combinations. So the above asset declaration would essentially be expanded into:

```
#asset(qx/icon/Tango/16/*)
#asset(qx/icon/Tango/32/*)
#asset(qx/icon/Oxygen/16/*)
#asset(qx/icon/Oxygen/32/*)
```

"library" Key and Manifest Files

The *library* key of a configuration holds information about source locations that will be considered in a job (much like the CLASSPATH in Java). Each element specifies one such library. The term "library" is meant here in the broadest sense; everything that has a qooxdoo application structure with a *Manifest.json* file can be considered a library in this context. This includes applications like the Showcase or the Feedreader, add-ins like the Testrunner or the Apiviewer, contribs from the qooxdoo-contrib repository, or of course the qooxdoo framework library itself. The main purpose of any *library* entry in the configuration is to provide the path to the library's "Manifest" file.

Manifest files

Manifest files serve to provide meta information for a library in a structured way. Their syntax is again JSON, and part of them is read by the generator, particularly the `provides` section. See *here* for more information about manifest files.

Contrib libraries

Contributions can be included in a configuration like any other libraries: You add an appropriate entry in the `library` array of your configuration. Like other libraries, the contribution must provide a *Manifest.json* file with appropriate contents.

If the contribution resides on your local file system, there is actually no difference to any other library. Specify the relative path to its Manifest file and you're basically set. The really new part comes when the contribution resides online, in the qooxdoo-contrib repository. Then you use a special syntax to specify the location of the Manifest file. It is URL-like with a `contrib` scheme and will usually look like this:

```
contrib://<ContributionName>/<Version>/<ManifestFile>
```

The contribution source tree will then be downloaded from the repository, the generator will adjust to the local path, and the contribution is then used just like a local library. A consideration that comes into play here is where the files are placed locally. The default location is a subdirectory from your cache path named `downloads`. You can modify this through the *downloads* attribute of the *cache* key in your config.

So, for example an entry for the "trunk" version of the "Dialog" contribution would look like this:

```
{
  "manifest" : "contrib://Dialog/trunk/Manifest.json"
}
```

You will rarely need to set the `uri` attribute of a library entry. This is only necessary if the relative path to the library (which is automatically calculated) does not represent a valid URL path when running the **source** version of the final app. (This can be the case if your try to run the source version from a web server that requires you to set up different document roots). It is not relevant for the *build* version of your app, as here all resources from the various libraries are collected under a common directory. For more on URI handling, see the next section.

"contrib://" URIs and Internet Access As contrib libraries are downloaded from an online repository, you need Internet access to use them. Here are some tips on how to address offline usage and Internet proxies.

Avoiding Online Access If you need to work with a contrib offline, it is best to download it to your hard disk, and then use it like any local qooxdoo library. Sourceforge offers the "ViewVC" online repository browser, so you can browse the contrib online, e.g.

```
http://qooxdoo-contrib.svn.sourceforge.net/viewvc/qooxdoo-contrib/trunk/qooxdoo-contrib/Dialog/
```

Browse to the desired contrib version, like *trunk*, and hit the *"Download GNU tarball"* link. This will download an archive of this part of the repository tree. Unpack it to a local directory, and enter the relative path to it in the corresponding *manifest* config entry. Now you are using the contrib like a local library.

The only thing you are missing this way is the automatic online check for updates, where a newer version of the contrib would be detected and downloaded. You need to do this by hand, re-checking the repository when you can, and re-downloading a newer version if you find one.

Accessing Online from behind a Proxy If you are sitting behind a proxy, here is what you can do. The generator uses the *urllib* module of Python to access web-based resources. This module honors proxies:

- It checks for a *http_proxy* environment variable in the shell running the generator. On Bash-like shells you can set it like this:

  ```
  http_proxy="http://www.someproxy.com:3128"; export http_proxy
  ```

- If there is no such shell setting on Windows, the registry is queried for the Internet Options.
- On MacOS, the Internet Config is queried in this case.
- See the module documentation for more details.

URI handling

URIs are used in a qooxdoo application to refer from one part to other parts like resources. There are places within the generator configuration where you can specify *uri* parameters. What they mean and how this all connects is explained in this section.

Where URIs are used The first important thing to note is:

Note: All URI handling within qooxdoo is related to libraries.

Within qooxdoo the *library* is a fundamental concept, and libraries in this sense contain all the things you are able to include in the final Web application, such as class files (.js), graphics (.gif, .png, ...), static HTML pages (.htm, .html), style sheets (.css), and translation files (.po).

But not all of the above resource types are actually referenced through URIs in the application. Among those that are you find in the **source** version:

- references to class files
- references to graphics
- references to static HTML
- references to style sheet files

The build version uses a different approach, since it strives to be a self-contained Web application that has no outgoing referenc

- JS class code is put into the (probably various) output files of the generator run (what you typically find under the *build/script* path). The bootstrap file references the others with relative URIs.
- Graphics and other resources are referenced with relative URIs from the compiled scripts. Those resources are typically found under the *build/resource* path.
- Translation strings and CLDR information can be directly included in the generated files (where they need not be referenced through URIs), or be put in separate files (where they have to be referenced).

So, in summary, in the *build* version some references might be resolved by directly including the specific information, while the remaining references are usually confined to the build directory tree. That is why you can just pack it up and copy it to your web server for deployment. The *source* version is normally used directly off of the file system, and employs relative URIs to reference all necessary files. Only in cases where you e.g. need to include interaction with a backend you will want to run the source version from a web server environment. For those cases the following details will be especially interesting. Others might want to skip the remainder of this section for now.

Although the scope and relevance of URIs vary between *source* and *build* versions, the underlying mechanisms are the same in both cases, with the special twist that when creating the *build* version there is only a single "library" considered, the build tree itself, which suffices to get all the URIs out fine. These mechanisms are described next.

Construction of URIs through the Generator So how does the generator create all of those URIs in the final application code? All those URIs are constructed through the following three components:

```
to_libraryroot [1]  + library_internal_path [2] + resource_path [3]
```

So for example a graphics file in the qooxdoo framework might get referenced using the following components

- [1] *"../../qooxdoo-1.3.1-sdk/framework/"*
- [2] *"source/resource/qx/"*
- [3] *"static/blank.gif"*

to produce the final URI *"../../qooxdoo-1.3.1-sdk/framework/source/resource/qx/static/ blank.gif"*.

These general parts have the following meaning:

- **[1]** : URI path to the library root (as will be valid when running the app in the browser). If you specify the *uri* parameter of the library's entry in your config, this is what gets used here.
- **[2]** : Path segment within the specific library. This is taken from the library's *Manifest.json*. The consumer of the library has no influence on it.
- **[3]** : Path segment leading to the specific resource. This is the path of the resource as found under the library's resource directory.

Library base URIs in the Source version Part *[1]* is exactly what you specify with the *uri* subkey of an entry in the *library* key list. All *source* jobs of the generator using this library will be using this URI prefix to reference resources of that library. (This is usually fine, as long as you don't have different autonomous parts in your application using the same library from different directories; see also further down).

If you don't specifying the *uri* key with your libraries (which is usually the case), the generator will calculate a value for *[1]*, using the following information:

```
applicationroot_to_configdir [1.1] + configdir_to_libraryroot [1.2]
```

The parts have the following meaning:

- **[1.1]** : Path from the Web application's root to the configuration file's directory; this information is derived from the *paths/app-root* key of the *compile-options* config key.
- **[1.2]** : Path from the configuration file's directory to the root directory of the library (the one containing the *Manifest.json* file); this information is immediately available from the library's *manifest* key.

For the **build** version, dedicated keys *uris/script* and *uris/resource* are available (as there is virtually only one "library"). The values of both keys cover the scope of components [1] + [2] in the first figure.

Since [1.2] is always known (otherwise the whole library would not be found), only [1.1] has to be given in the config. The properties of this approach, compared to specifying just [1], are:

- *The application root can be specified individually for each compile job.* This means you could have more than one application root in your project, e.g. when your main application offers an iframe, into which another application from the same project is loaded; qooxdoo's Demobrowser application takes advantage of exactly this.

- *Relative file system paths have to match with relative URIs in the running application.* So this approach won't work if e.g the relative path from your config directory to the library makes no sense when the app is run from a web server.

From the above discussion, there is one important point to take away, in order to create working URIs in your application:

Note: You have to either specify the library's *uri* parameter ([1]) or the URI-relevant keys in the compile jobs ([1.1]) in your config.

While either are optional in their respective contexts, it is mandatory to at least specify *one* of them for the URI generation to work.

Overriding the 'uri' settings of libraries Libraries you specify in your own config (with the *library* key) are in your hand, and you can provide `uri` parameters as you see fit. If you want to tweak the *uri* setting of a library entry that is added by including another config file (e.g. the default *application.json*), you simply re-define the library entry of that particular library locally. The generator will realize that both entries refer to the same library, and your local settings will take precedence.

Specifying a "library" key in your config.json

You can specify `library` keys in your own config in these ways:

- You either define a local job which either shaddows or "extends" an imported job, and provide this local job with a `library` key. Or,
- You define a local `"libraries"` job and provide it with a `library` key. This job will be used automatically by most of the standard jobs (source, build, etc.), and thus your listed libraries will be used in multiple jobs (not just one as above).

"packages" Key

For a general introduction to parts and packages see this separate *document*. Following here is more information on the specifics of some sub-keys of the *packages* config key.

parts/<part_name>/include

The way the part system is currently implemented has some caveats in the way *parts/*/include* keys and the general *include* key interact:

1. The general *include* key provides the "master list" of classes for the given application. This master list is extended with all their recursive dependencies. All classes given in a part's *include*, including all their dependencies, are checked against this list. If any of those classes is not in the master list, it will not be included in the final app.

   Therefore, you cannot include classes in parts that are not covered by the general *include* key. If you want to use e.g. *qx.bom.** in a part, you have to add `"qx.bom.*"` to the general *include* list. Otherwise, only classes within *qx.bom.** that actually derive from the general include key will be actually included, and the rest will be discarded. Motto:

   "The general include key is a filter for all classes in parts."

2. Any class that is in the master list that is never listed in one of the parts, either directly or as dependency, will not be included in the app. That means you have to **actively** make sure that all classes from the general *include* get - directly or indirectly - referenced in one of the parts, or they will not be in the final app. Motto:

"The parts' include keys are a filter for all classes in the general include key."

Or, to put both aspects in a single statement: The classes in the final app are exactly those in the **intersection** of the classes referenced through the general *include* key and all the classes referenced by the *include* keys of the parts. Currently, the application developer has to make sure that they match, ie. that the classes specified through the parts together sum up to the global class list!

There is another caveat that concerns the relation between *include*'s of different parts:

3. Any class that is listed in a part's *include* (file globs expanded) will not be included in another part. - But this also means that if two parts list the same class, it won't be included in either of them!

   This is e.g. the case in a sample application, where the *boot* part lists *qx.bom.client.Engine* and the *core* part lists *qx.bom.** which also expands to *qx.bom.client.Engine* eventually. That's the reason why *qx.bom.client.Engine* would not be contained in either of those parts, and hence would not be contained in the final application at all.

i18n-with-boot

Setting this sub-key to *false* will result in I18N information (translations, CLDR data, ...) being put in their own separate parts. The utility of this is:

- The code packages get smaller, which allows for faster application startup.
- You can handle I18N data more individually.

Here are the details:

- By default, I18N data, i.e. translations from the .po files and CLDR data, is integrated as Javascript data in the application packages. In the source version, where there are no generated packages, this data is integrated with the loader. Setting *packages/i18n-with-boot: false* removes this data from the code packages (loader or otherwise; don't think too much about the key name).
- Rather, data for *each individual locale* (en, en_US, de, de_DE, ...) will be collated in a dedicated *part*, the part name being that of the respective language code. As usual, each part is made up of packages. In the case of an I18N part, these are the corresponding data package plus fall-back packages for key lookup (e.g. ["C", "en", "en_US"] for the part *en_US*). Each package is a normal qooxdoo package with only the data section, and without the code section.

So far, so good. This is the point where the application developer has to take over. The application will not load the I18N parts by itself. You have to do it using the usual part loading API (e.g. `qx.io.PartLoader.require(["en_US"])`). You might want to do that early in the application run cycle, e.g. during application start-up and before the first translateable string or localizable data is to be displayed. After loading the part, the corresponding locale is ready to be used in the normal way in your application. The Feedreader application uses this technique to load a different I18N part when the language is changed in its *Preferences* dialog.

"include" Key (top-level) - Adding Features

Within qooxdoo there are a couple of features that are not so much applications although they share a lot of the classical application structure. The APIViewer and TestRunner are good examples for those. (In the recent repository re-org, they have been filed under *component* correspondingly). They are applications but receive their actual meaning from other applications: An APIViewer in the form of class documentation it presents, the TestRunner in the form of providing an environment to other application's test classes. On their own, both applications are "empty", and the goal is it to use them in the context of another, self-contained application. The old build system supported make targets like 'api' and 'test' to that end.

While you can always include other applications' *classes* in your project (by adding an entry for them to the *library* key of your config), you wouldn't want to repeat all the necessary job entries to actually build this external app in your environment. So the issue here is not to re-use classes, but *jobs*.

Re-using jobs

So, the general issue we want to solve is to import entire job definitions in our local configuration. The next step is then to make them work in the local environment (e.g. classes have to be compiled and resources be copied to local folders). This concepts is fairly general and scales from small jobs (where you just keep their definition centrally, in order to use them in multiple places) to really big jobs (like e.g. creating a customized build version of the Apiviewer in your local project).

Practically, there are two steps involved in using external jobs:

1. You have to *include* the external configuration file that contains the relevant job definitions. Do so will result in the external jobs being added to the list of jobs of your local configuration. E.g. you can use

   ```
   generator.py ?
   ```

   to get a list of all available jobs; the external jobs will be among this list.

2. There are now two way to utilize these jobs:

- You can either invoke them directly from the command line, passing them as arguments to the generator.
- Or you define local jobs that *extend* them.

In the former case the only way to influence the behaviour of the external job is through macros: The external job has to parameterize its workings with macro references, you have to know them and provide values for them that are suitable for your environment (A typical example would be output paths that you need to customize). Your values will take precendence over any values that might be defined in the external config. But this also means you will have to know the job, know the macros it uses, provide values for them (e.g. in the global *let* of your config), resolve conflicts if other jobs happen to use the same macros, and so forth.

In the latter case, you have more control over the settings of the external job that you are actually using. Here as well, you can provide macro definitions that parameterize the behaviour of the job you are extending. But you can also supply more job keys that will either shaddow the keys of the same name in the external job, or will be extended by them. In any case you will have more control over the effects of the external job.

Add-ins use exactly these mechanisms to provide their functionality to other applications (in the sense as 'make test' or 'make api' did it in the old system). Consequently, to support this in the new system, the add-in applications (or more precisely: their job configuration) have to expose certain keys and use certain macros that can both be overridden by the using application. The next sections describe these build interfaces for the various add-in apps. But first more practical detail about the outlined ...

Add-In Protocol

In order to include an add-in feature in an existing app, you first have to `include` its job config. On the top-level of the config map, e.g. specify to include the Apiviewer config:

```
"include" : [{"path": "../apiviewer/config.json"}]
```

The include key on this level takes an array of maps. Each map specifies one configuration file to include. The only mandator key therein is the file path to the external config file (see *here* for all the gory details). A config can only include what the external config is willing to *export*. Among those jobs the importing config can select (through the `import` key) or reject (through the `block` key) certain jobs. The resulting list of external job definitions will be added to the local jobs map.

If you want to fine-tune the behaviour of such an imported job, you define a local job that extends it. Imported jobs are referenced like any job in the current config, either by their plain name (the default), or, if you specify the `as` key in the include, by a composite name `<as_value>::<original_name>`. Suppose you used an `"as" :`

`"apiconf"` in your include, and you wanted to extend the Apiviewer's `build-script` job, this could look like this:

```
"myapi-script" :
{
  "extend" : ["apiconf::build-script"]
  ...
}
```

As a third step, the local job will usually have to provide additional information for the external job to succeed. Which exactly these are depends on the add-in (and should eventually be documented there). See the section specific to the *APIViewer* for a concrete example.

API Viewer

For brevity, let's jump right in into a config fragment that has all necessary ingredients. These are explained in more detail afterwards.

```
{
  "include" : [{"as" : "apiconf", "path" : "../apiviewer/config.json"}],
  "jobs" : {
    "myapi" : {
        "extend" : ["apiconf::build"],
        "let" : {
            "ROOT"   :  "../apiviewer",
            "BUILD_PATH" : "./api",
            "API_INCLUDE" : ["qx.*", "myapp.*"],
            "API_EXCLUDE" : ["myapp.tests.*"]
            },
        "library" : { ... },
        "settings" : {
            "myapp.resourceUri" : "./resource"
            }
        }
    }
}
```

The `myapi` job extends the `build` job of APIViewer's job config. This "build" job is itself a run job, i.e. it will be expanded in so many individual jobs as its `run` key lists. All those jobs will get the "myapi" job as a context into which they are expanded, so all other settings in "myapi" will be effective in those jobs.

In the `let` key, the ROOT, BUILD_PATH, API_INCLUDE and API_EXCLUDE macros of the APIViewer config are overridden. This ensures the APIViewer classes are found, can be processed, and the resulting script is put into a local directory. Furthermore, the right classes are included in the documentation data.

The `library` key has to at least add the entry for the current application, since this is relevant for the generation of the api documentation for the local classes.

So in short, the `ROOT`, `BUILD_PATH`, `API_INCLUDE` and `API_EXCLUDE` macros define the interface between the apiviewer's "run" job and the local config.

"optimize" Key

The *optimize* key is a subkey of the *compile-options key*. It allows you to tailor the forms of code optimization that is applied to the Javascript code when the *build* version is created. The best way to set this key is by setting the *OPTIMIZE macro* in your config's global *let* section. Currently, there are four categories which can be optimized.

strings With string optimization, strings are extracted from the class definition and put into lexical variables. The occurrences of the strings in the class definition is then replaced by the variable name. This mainly benefits IE6 and repetitive references to the same string literal.

variables Long variable names are made short. Lexical variables (those declared with a *var* statement) are replaced by generated names that are much shorter (1-2 characters on average). Dependending on the original code, this can result in significant space savings.

privates This is less an optimization in space or time, but rather a way to enforce privates. Private members of a class (those beginning with "__") are replaced with generated names, and are substituted throughout the class. If some other class is accessing those privates, these references are not updated and will eventually fail when the access happens. This will lead to a runtime error.

basecalls Calls to *this.base()*, which invoke the corresponding superclass method, are inlined, i.e. the superclass method call is inserted in place of the this.base() call.

8.3 Further Tools

8.3.1 Source Code Validation

qooxdoo includes its own Javascript validator, **Ecmalint**, which application developers can use to check their source files for errors. It is started by running the *lint* generator job in an application directory:

```
./generate.py lint
```

Critical Warnings

Use of undefined or global identifier

This warning indicates that an unknown global variable is used. This can be caused by:

- The variable is not declared as local variable using `var`
- The variable name is misspelled
- It is OK to use this global but EcmaLint does not know about it. This can be fixed by passing the variable name as known variable to the EcmaLint call or by adding a `@lint ignoreUndefined(VARIABLE_NAME)` doc comment to the method's API doc comment

Unused identifier

Map key redefined

Data field has a reference value

Hint: If data fields are initialized in the members map with reference values like arrays or maps they will be shared between all instances of the class. Usually it is better to set the value to 'null' and initialize it in the constructor

Use of deprecated identifier

Critical Warning (for framework)

Potentially non-local private data field

Hint: You should never do this.

Protected data field

Hint: Protected data fields are deprecated. Better use private fields in combination with getter and setter methods.

Comment: It appears that this isn't an issue that is generically to be solved as the hint suggest. See the corresponding bug report.

Undeclared private data field

Hint: You should list this field in the members section.

Coding Style Warnings

The statement of loops and conditions must be enclosed by a block in braces

Multiply declared identifier

Explicitly ignoring messages

Starting with qooxdoo 0.8.3 the following three doc comments can be used to explicitly ignore specific lint messages:

```
@lint ignoreUnused(x, y)
@lint ignoreDeprecated(alert)
@lint ignoreUndefined(button1, foo)
@lint ignoreReferenceField(field)
```

Before lint prints a warning it walks up the AST and looks for the next enclosing API doc comment. Usually these comments should be placed in method JsDoc comments or in the class comment.

Suppressing additional warnings is not supported because they are always an error (e.g. duplicate map keys) or are very hard to implement (e.g. protected warnings).

8.4 Specific Topics

8.4.1 Code Compilation

ASTlets - AST Fragments

Note: Work in Progress

This is an ongoing page to record and document the AST (abstract syntax tree) fragments (*"ASTlets"*), as they are generated by the tool chain Javascript parser. It shows how certain JS syntax constructs get translated into the corresponding AST representation. This serves mainly internal purposes and should not be relevant for a qooxdoo application developer.

The notation is a simplified tree structure that names token symbols and their nesting through indentation. "|" denotes alternatives.

Syntax Constructs

a[i]

```
accessor
  identifier ("a")
  key
    variable
      identifier ("i")
```

a()

```
call
  operand
    variable
      identifier ("a")
  params
```

{a : 1}

```
map
  keyvalue ("a")
    value
      constant (1)
```

a = b

```
assignment
  left
    variable
      identifier ("a")
  right
    variable
      identifier("b")
```

a.b.c(d)

```
call
  operand
    variable
      identifier ("a")
      identifier ("b")
      identifier ("c")
  params
    variable
      identifier ("d")
```

a.b().c(d)

```
accessor
  left
    call
      operand
        variable
          identifier ("a")
          identifier ("b")
  right
    call
      operand
        variable
          identifier ("c")
      params
        variable
          identifier ("d")
```

[file:] a.b("c",{d:e})

```
file
  call
    operand
      variable
        identifier ("a")
        identifier ("b")
    params
      constant  ("c")
      map
        keyvalue ("d")
          value
            variable
              identifier ("e")
```

(function () {return 3;})() *(anonymous function immediately called)*

```
call
  operand
    group
      function
        params
        body
          block
            return
              expression
                constant ("3")
```

function () {return 3;}() *(anonymous function immediately called - no paren)*

```
call
  operand
    function
      params
      body
        block
          return
```

```
expression
  constant ("3")
```

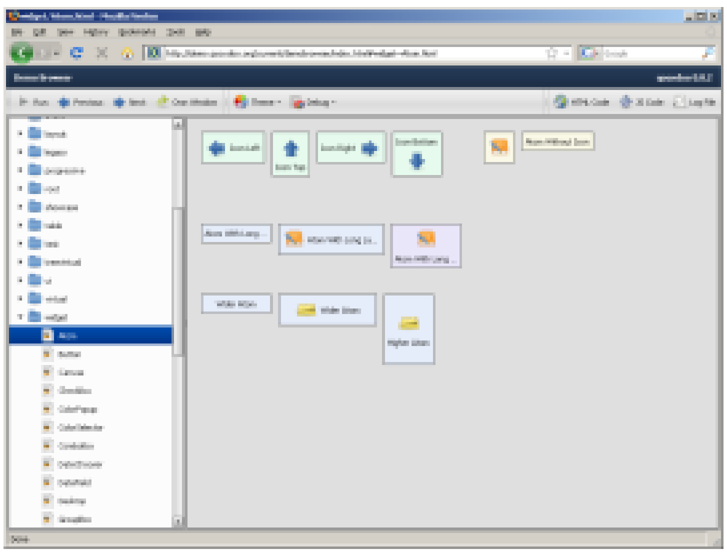
CHAPTER

NINE

STANDARD APPLICATIONS

9.1 Demo Applications

9.1.1 Demobrowser

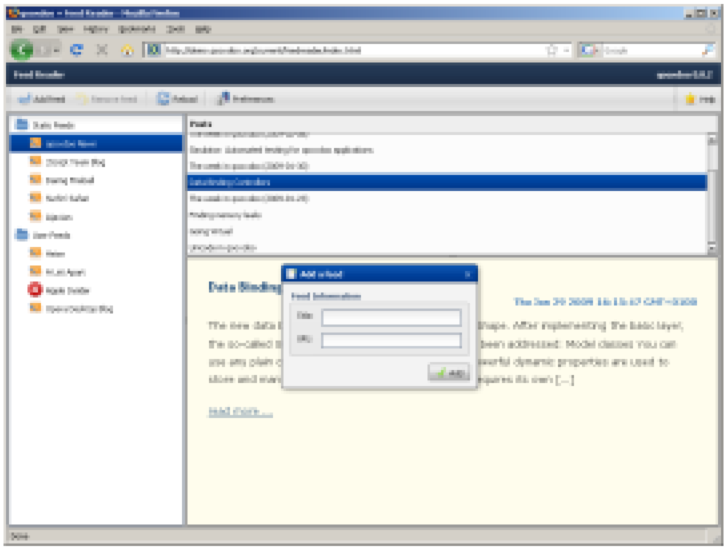


Demobrowser is a runner frame application that hosts nearly 250 sample applications. All those applications are full, stand-alone qooxdoo applications. They cover a wide range, from simple layout test apps to more full-featured form applications. But the focus is usually to exercise a particular widget or feature of the qooxdoo class library. So while they might be simple visually, they are good resources to see how a specific feature can be used in an app. As they are generated in two variants, one for each of qooxdoo's standard themes, you can get an impression how the same application looks under the different themes.

To navigate these demo applications, the runner frame organizes them in a navigation tree that groups applications by main feature (like data binding, layouts, events, etc.). It also allows you to search for demo titles and qooxdoo classes used in the demos, so you can e.g. search for all demos that deploy qooxdoo's List widget (qx.ui.form.List). For each rendered app you can inspect the JavaScript source to see how it is done.

> Contains many examples and tests for widgets, layouts and other framework functionality.
>
> Online demo

9.1.2 Feedreader

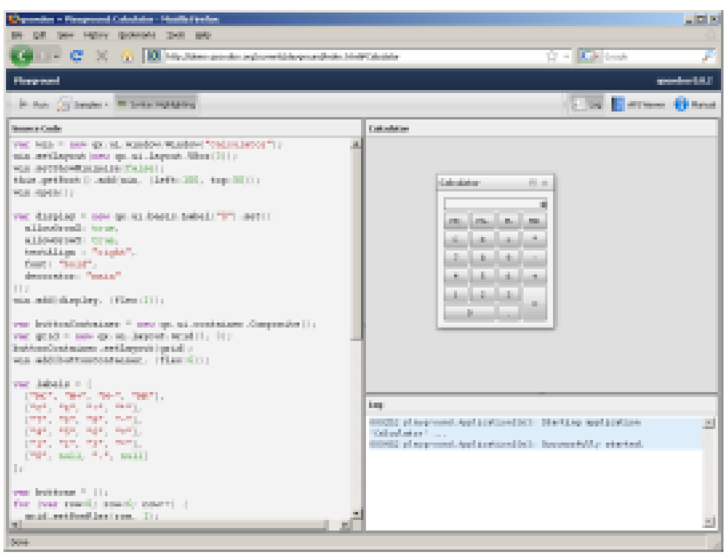


Feedreader is a browser-based RSS feedreader. It allows you to read posts of pre-defined feeds, but you can also add other feeds in a session. The indiviudal feeds are retrieved using YQL queries. It also showcases switching the language for an application, offering seven languages to choose from. As it uses internet access, internationalization and is organized in parts on the code level, it shows several features of prototypical RIA applications.

A typical rich internet application (RIA) for displaying RSS feeds.

Online demo

9.1.3 Playground



The Playground application allows you to play with code in an edit pane, and see the result of running that code in a preview pane. It comes with a set of pre-defined code samples, but many more are available, e.g. from github. Code can also be bookmarked and the links saved and re-run, to re-create the sample you were working on. This allows for easy sharing of running code samples with others.

The scope of the code you can enter in the edit pane is restricted to what you can do in the `main()` method of a standard qooxdoo application class.

Bookmarklet

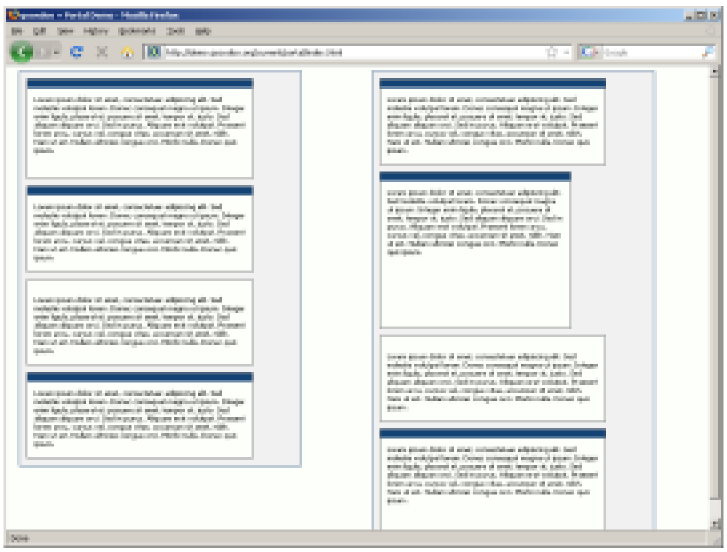
Note: experimental

```
javascript:(function(s){try{s=document.selection.createRange().text}catch(e){s=window.getSelection()
```

Explore qooxdoo programming interactively: edit qooxdoo code in one pane, and see the result running in another.

Online demo

9.1.4 Portal



Portal is a DOM-level application that doesn't use any of qooxdoo's GUI widgets. It shows both what you can do using qooxdoo's low-level API and how to build a rudimentary portal application. The various portlets can be freely re-arranged by dragging them to new positions. They also indicate how they can be resized.

A low-level, DOM-oriented application without any high-level qooxdoo widgets.

Online demo

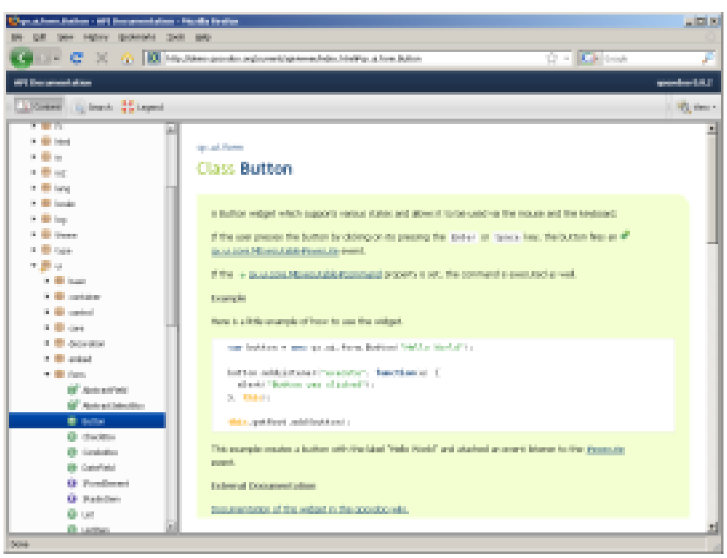
9.1.5 Showcase

The Showcase application contains a number of feature "applets" that are actually parts of a single qooxdoo application. A thumbnails bar allows you to switch between the different demo apps. The topics covered include tabling, themes, internationalization and data binding. With each app there comes instructions and background information, how to exercise them, links to further demos and documentation.

> A page-style application embedding a number of small showcase applications to highlight specific topics like tables, theming or internationalization.
>
> Online demo

9.2 Developer Tools

9.2.1 Apiviewer



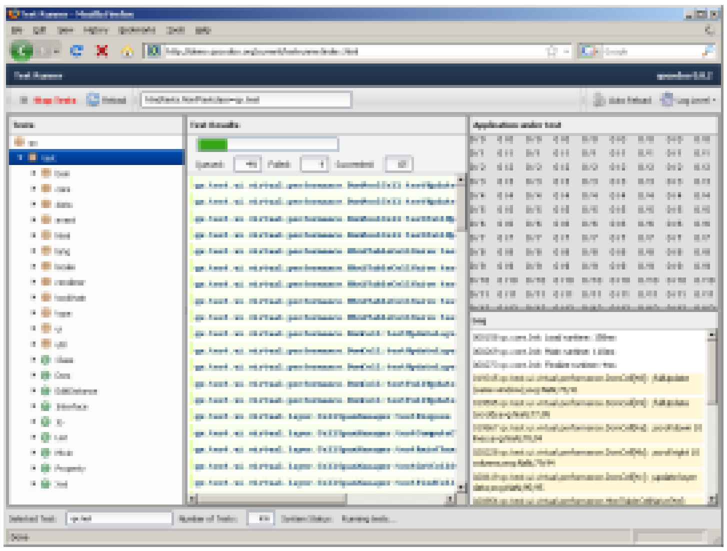
The Apiviewer is an application to browse qooxdoo's class API. The tree view pane offers the typical class hierarchy, organized by name spaces. Each package (intermediate name space) has an overview description and links to the sub-packages or classes it contains. Descriptions usually contain cross links to relevant packages or classes. The entire reference is searchable through the search tab, where you can enter class and method names.

The actual API descriptions are generated from JSDoc-style comments in the class code, and can be generated for custom applications as well, so you can browse the API of your own classes in Apiviewer.

Searchable API reference of the qooxdoo framework.

Online demo

9.2.2 Testrunner



Testrunner is a runner frame application for unit tests. Unit tests can be written using framework classes from the `qx.dev.unit.*` name space. They follow the general scheme of JSUnit. Test class are then gathered into a dedicated test application (the "Application under Test"). This test application is loaded into the runner frame, which discovers and displays the contained tests, and allows you to run them all or individually. Results and log entries are displayed in dedicated panes.

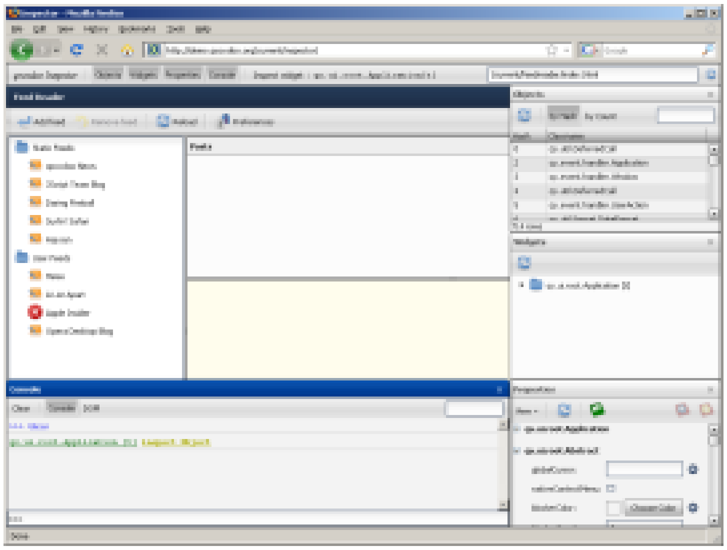
The online Testrunner loads qooxdoo's own framework unit tests (approx. 1,500 unit tests currently), but a custom testrunner can be created for each custom application, to run that application's unit tests.

Integrated unit testing framework similar to (but not requiring) JSUnit.

Online demo

9.2.3 Inspector

qooxdoo Inspector is a powerful development tool perfectly suited for *live* debugging and modifying qooxdoo applications.



See it in action, debugging the qooxdoo feedreader demo application.

If you know the Firebug extension for Firefox, you will be familiar with most of Inspector's capabilities. But it is much more than that: Since it is a qooxdoo application itself, it runs in *all* major browsers, including IE, Firefox, Opera, Safari and Chrome. And it allows for truly *qooxdoo-specific* debugging, including displaying the UI hierarchy and modifying the properties of qooxdoo widgets.

Usage

There are two ways to use the inspector (explained in more detail below):

- The first way is to run a simple generator job to create a local inspector instance for your custom application. (See: *Running the inspector job*)
- Generate the build version of the inspector and open it in a (local) web server. (See: *Running inspector with an HTTP server*)

Individual inspector from file system

First of all, make sure you've created a source version of your application. Then create the inspector:

```
generate.py inspector
```

Once the job is finished, you can open the `index.html` file from the created inspector application. You will find the file in the newly generated inspector folder (`inspector\index.html`).

Shared inspector over a web server

To generate the build version of the inspector, change to the inspector home directory (in the SDK in folder `SDK\component\inspector`). Then run its `build` job:

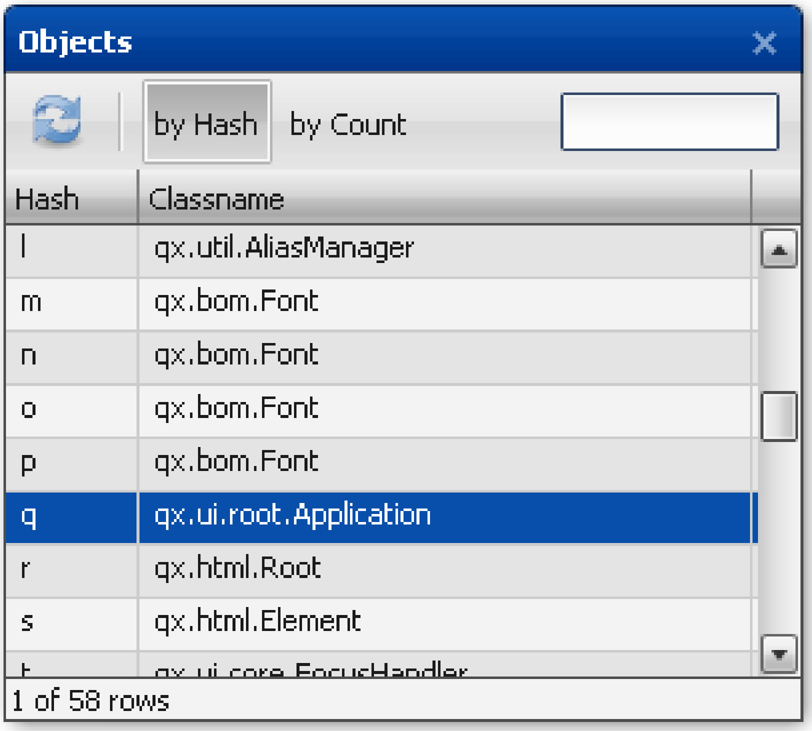
```
generate.py build
```

Once the build job is finished you can access the inspector through your HTTP server to inspect different qooxdoo applications. If you don't already have an HTTP server like Apache installed or you don't want to configure it, you can startup a simple Python-based web server locally:

```
python -m CGIHTTPServer
```

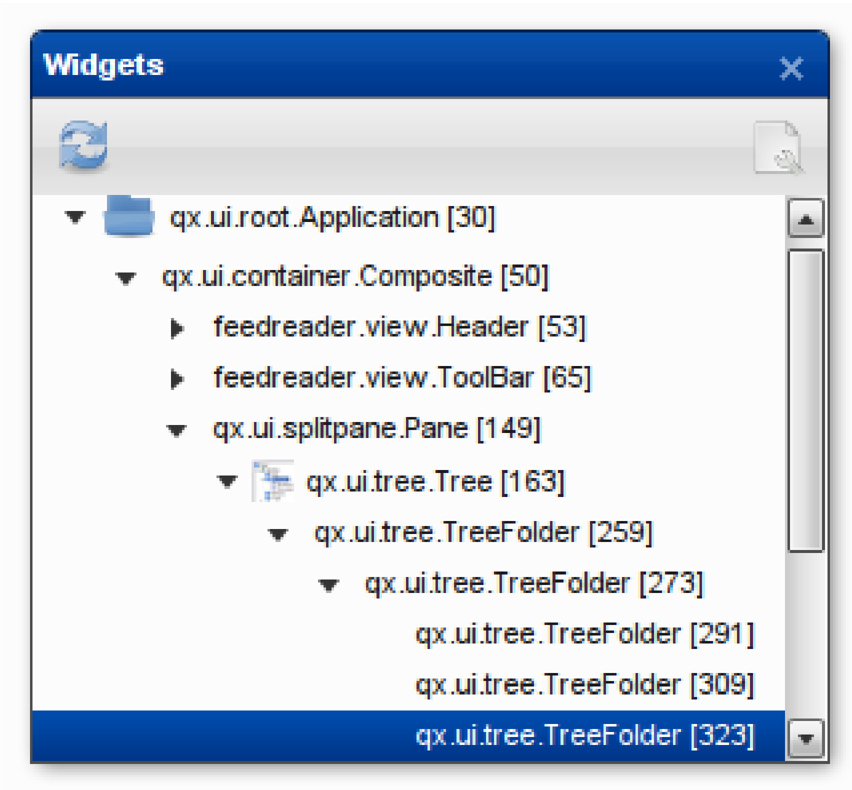
Note: Make sure the qooxdoo SDK (inspector) and the custom applications to debug are accessible from the document root! You can achieve this by starting up the Python command above from a directory that has both directories as subdirectories.

Objects Window



The objects window lists all qooxdoo objects created by your app in a table. The inspector has full access to the internal object registry of your application. Of course, the inspector's objects are excluded from the display so they won't interfere with debugging your app. The objects can be sorted by hash, count or name and filtered by name. To select an object listed in the table and to update the other views accordingly, simply click on its list entry.

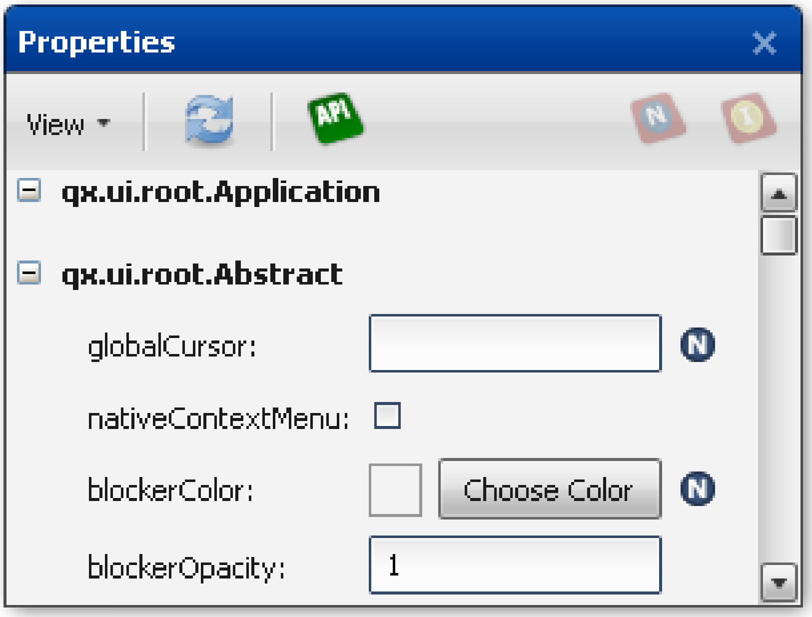
Widgets Window



The widgets window displays the hierarchical structure of your application's GUI as a tree. Each widget which was added to the document (or into any deeper widget hierarchy) will be shown. Again, a simple click on a widget in the tree selects it. Most of the widgets have a specific icon (corresponding to their type) in order to identify the widgets in the tree faster. The name of the widget's class and its hash value are shown as identifiers in the tree.

The widgets window has two display modes: By default, the application's "public" widget hierarchy is displayed, i.e. only those widgets that were explicitly added by the application developer using the parent widget's "add" method. Sub-widgets that are added by the parent widget itself ("child controls") are hidden in this mode. That's why it's possible to select a widget using the "Inspect widget" button or the Objects window without the Widgets tree displaying it. In that case, use the button in the top right corner to switch to the internal widget hierarchy display mode and click the "reload" button. After that, all sub-widgets including child controls will be displayed in the window.

Properties Window

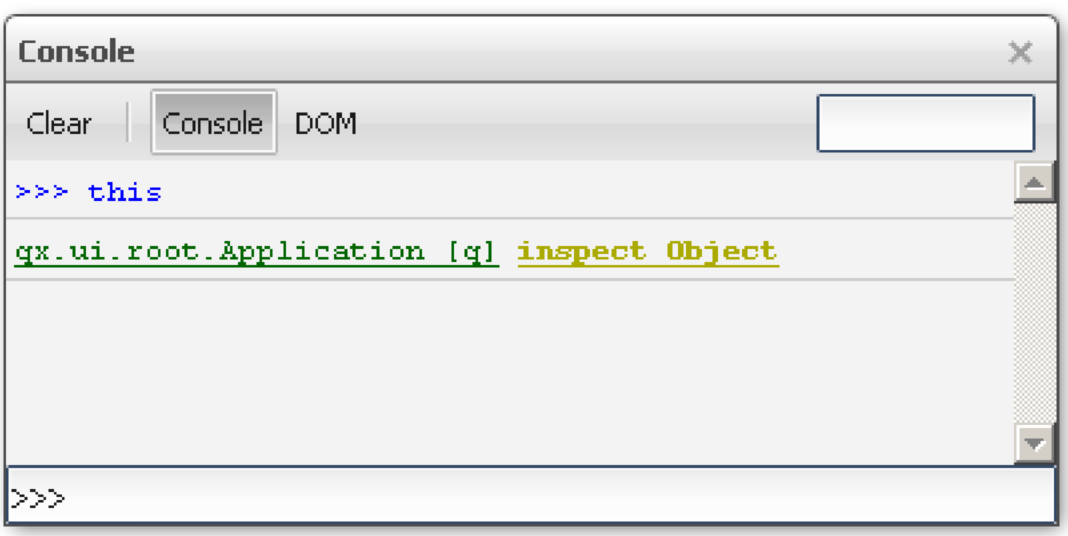


The properties window is one of two windows whose main focus is on actually *working* with a previously selected object. It shows all properties of the currently selected object. There are two different ways to sort the properties.

But it is not only about displaying properties, it also allows editing: To make this as convenient and least error-prone as possible, form elements are chosen according to the property's type. For instance, in many cases it is as easy as using a checkbox (for a boolean value), a drop-down menu (for pre-defined values) or a color picker (for a color value). For properties that support a wider range of values, regular text input fields are used.

If you want to know more about a certain property, select it and click the API button to open up the API documentation for the selected property.

Console

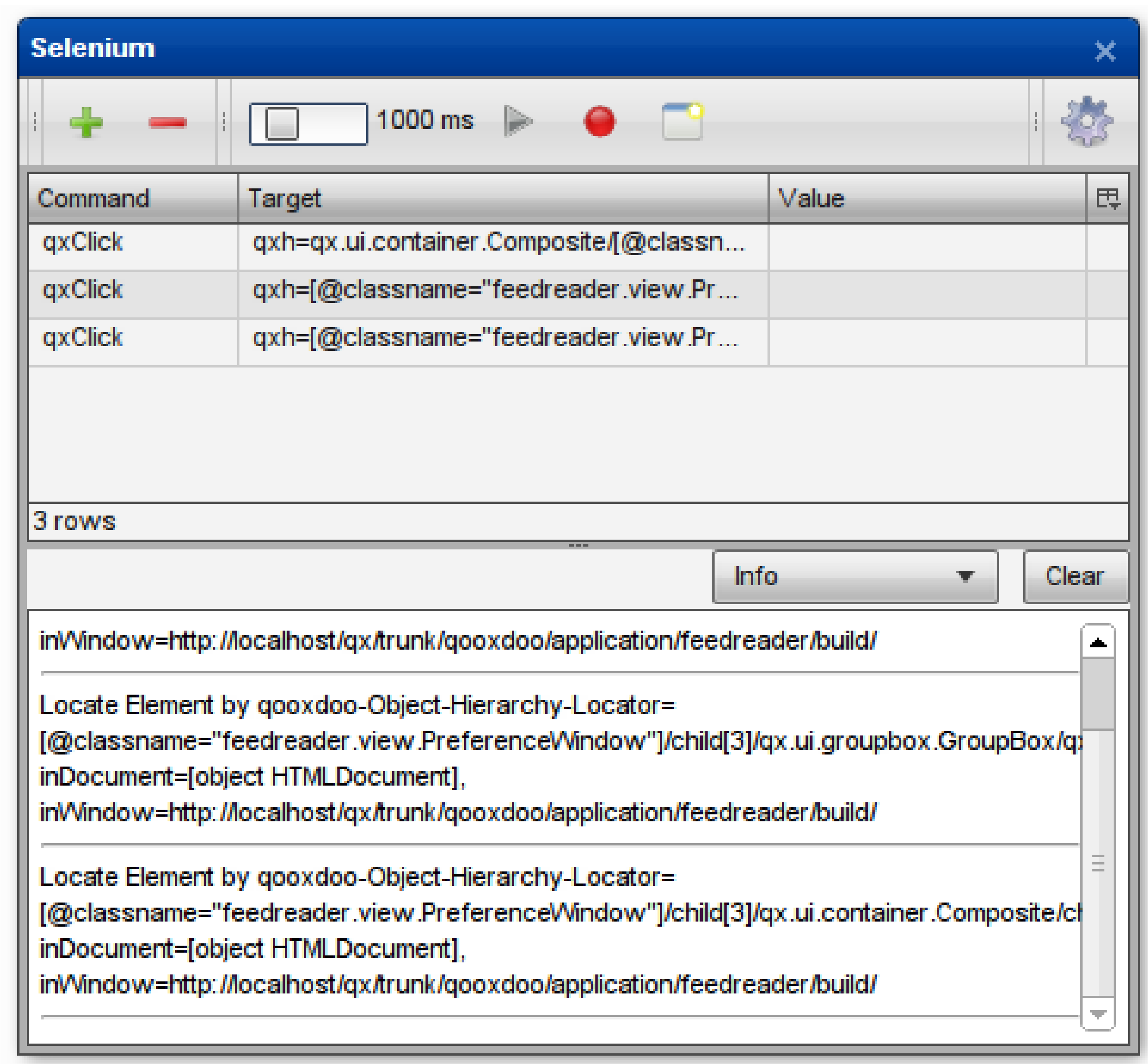


The console is probably the most powerful Inspector window, as it allows viewing and modifying instances similar to the properties window, but it also gives the developer a virtually unrestricted environment for debugging a qooxdoo app.

One part of the console is a generic JavaScript console, familiar to most Firebug users. At the prompt you can enter arbitrary JavaScript code which is executed after pressing enter. The keyword "this" refers to the currently selected object. That way it is very easy to inspect and modify the currently selected widget instance. To make it even more convenient, auto-completion while entering code is available. This allows you to select one of the suggested methods that are available for a specific object. Hit the CRTL+Space keys to display a list of available instance members.

Another part of the console window is a DOM browser, named as in Firebug. This browser allows you to inspect an object interactively. You can "dive into" an object, down to arbitrary depth, following property values that refer to data structures within the current object or pointing to ones within other objects.

Selenium Window



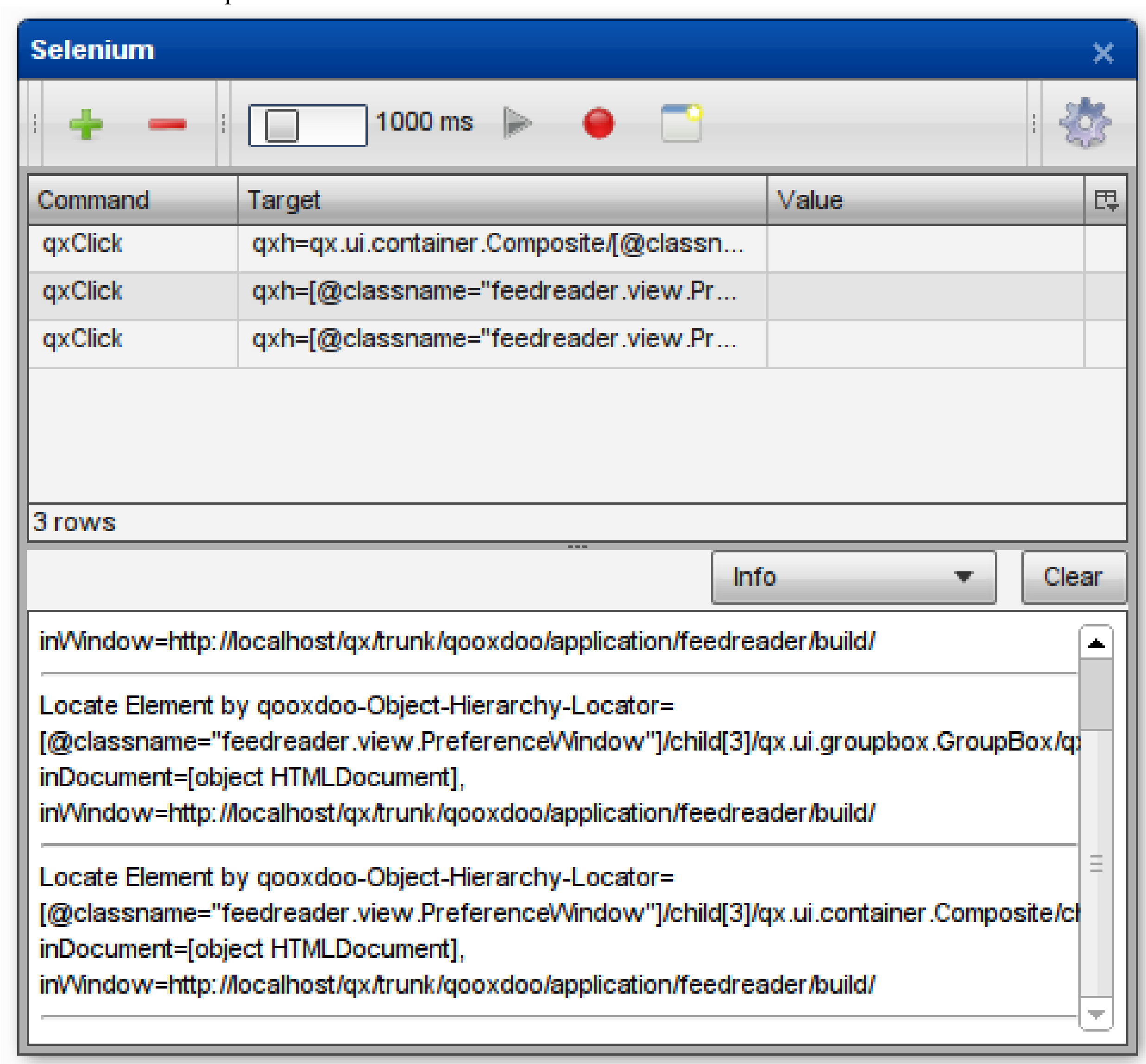
The Selenium window's purpose is to help test developers in writing simulated interaction tests which will then be run using the Selenium testing framework and qooxdoo's *Simulator component* or the Simulator contribution it is based on. Similar to the Selenium IDE Firefox plugin, it can be used to determine a locator string for any element (qooxdoo widget in this case) and supports playback of test commands against the inspected application.

There is a dedicated page with extensive descriptions that demonstrates how to create a test case using the Selenium window:

Using the Inspector to write Selenium tests

qooxdoo's Inspector is not only a very useful tool for application developers, it can also help you write Selenium tests.

The Selenium window From qooxdoo 1.2 onward, the Inspector features a Selenium window that duplicates some of the functionality of the Selenium IDE Firefox extension, with a qooxdoo twist. It can generate locator strings for any qooxdoo widget and run Selenium commands against the inspected application. The result is a simple Selenium test case that can be exported in the "Selenese" HTML format.



Prerequisites The Selenium window needs to load **Selenium Core** (the JavaScript part of Selenium) and the qooxdoo user extensions for Selenium to work. Selenium Core can be downloaded as a zip archive from the Selenium website. The user extensions are located in the qooxdoo framework since version 1.4 (`component/simulator/tool/user-extensions/user-extensions.js`). Users of older qooxdoo versions can use SourceForge's SVN view to download the file.

If the Inspector is loaded over HTTP, the required scripts can be loaded directly from their online repositories by clicking the `Set default URIs` button in the Options window.

Configuration Clicking the Options button (the only part of the Selenium window that is active initially) opens a window where these two settings can be defined. For Selenium Core, enter the URI of a directory where you've extracted the Selenium Core zip file.

The protocol used **must** be the same the Inspector is loaded over:

- If you're loading the Inspector from your local file system, extract the archive locally and use a file system URI (`file:///...`).
- If the Inspector is loaded from a web server, the Selenium Core directory must be accessed over HTTP. In this case, Same Origin Policy restrictions do **not** apply, so the script directory needn't be on the same server as the Inspector itself. If it is, a relative path can be used.

The same restrictions apply for the qooxdoo user extensions for Selenium, except here the path should just point to the one file. If the Inspector is accessed over HTTP, you can use this link to get the latest version directly from SVN:

`http://qooxdoo.svn.sourceforge.net/viewvc/qooxdoo/tags/release_1_4/qooxdoo/component/simulator/tool/`

Click "OK" after entering the paths. The rest of the Selenium window's GUI will be activated once the external scripts are loaded. Path information is saved in Cookies so this step is only necessary once per browser.

Controls

Toolbar

Pressing the **plus button** will add a new line to the test case. This consists of a default command (`qxClick`) and a `qxh` locator pointing to the widget currently selected in the Inspector.

The **minus button** removes the currently selected lines from the test case.

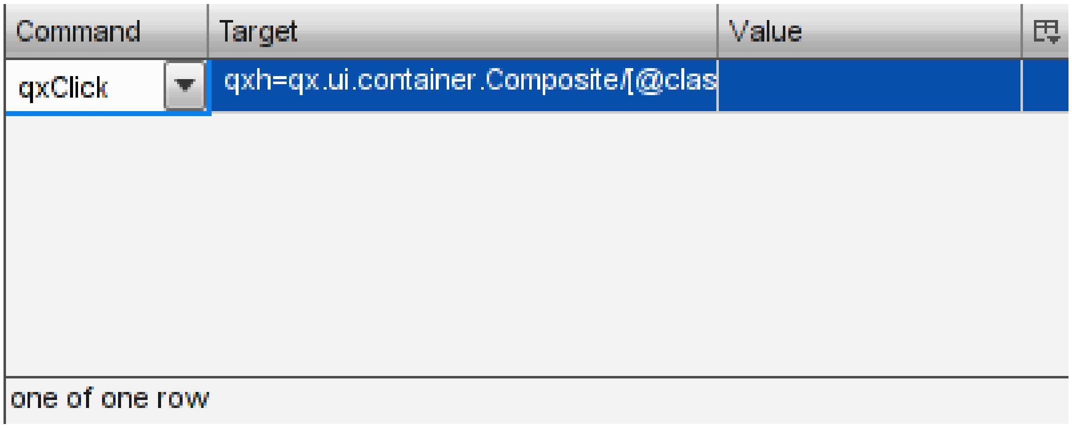
The **slider** controls the delay between individual commands when playing back a test case. In some cases, e.g. clicking a button that opens a new window, it will be necessary to set this to a higher value to make sure the application finishes rendering before the next command executes.

The **play button** executes selected test commands. If no commands are selected, the will all be run.

While the **record button** is active, a new line will be added whenever a new widget is selected in the Inspector.

The **import/export button** opens a new window containing the current test case in Selenese format. To import a Selenese test case, paste it into the text field and click `Import`.

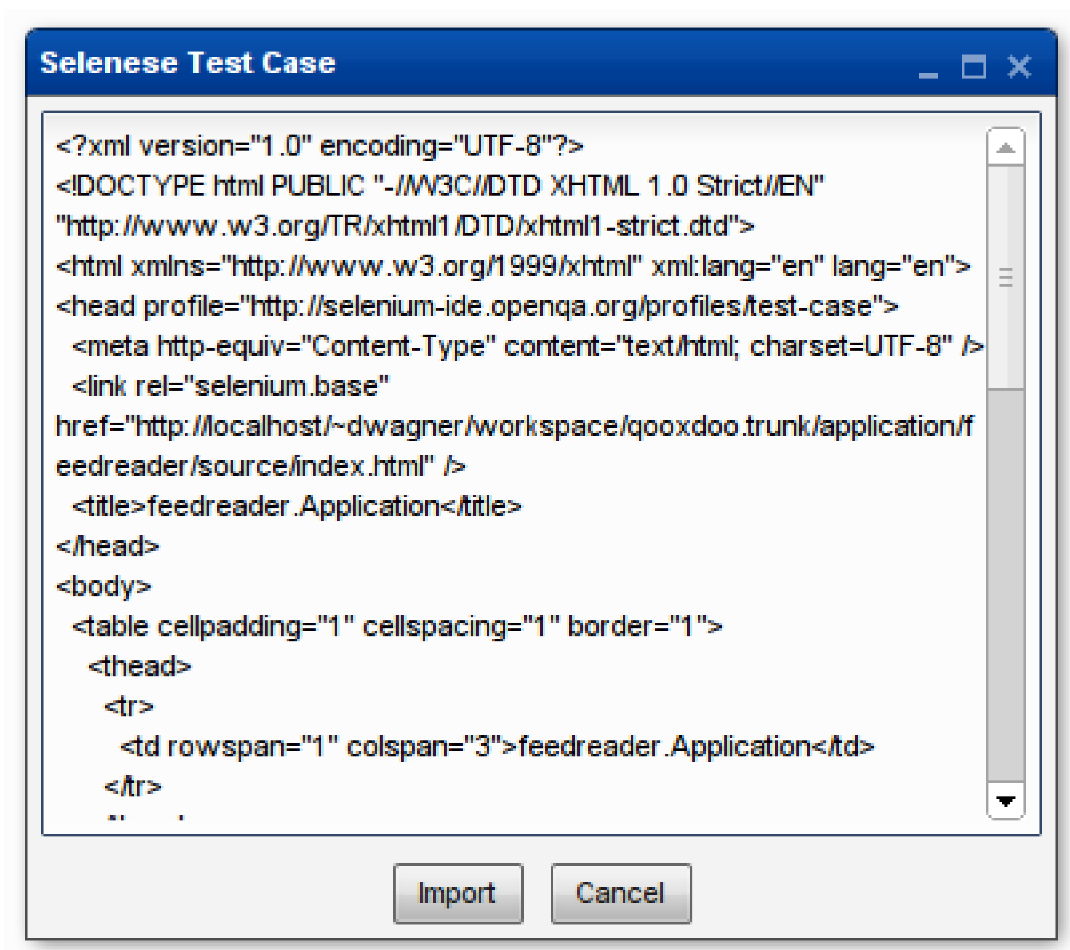
The **options button** opens a dialog where external script paths can be configured.



Test Commands

The table underneath the toolbar lists the commands in the current test case. Select one or more rows to execute their commands using the play button. Commands, locators and parameters can be edited by double clicking. Editing commands will display a combo box listing all commands supported by Selenium Core.

Log The log area displays any messages generated by Selenium Core while running commands.



Selenese window

Opened by clicking the Import/Export button in the toolbar, the Selenese window displays the current test case in Selenese format. This can be copied and pasted into a file, e.g. to be run by Selenium RC. Selenese import is also supported by pasting the contents of a Selenese file in the text area and clicking `Import`. This will replace any commands in the current test case with those from the pasted Selenese.

Tutorial To demonstrate the Selenium window, let's write a small test case for the qooxdoo Feed Reader: We'll automate the procedure of adding a new user-defined feed.

For this we'll need both the Feed Reader itself and the Inspector, of course: Generate both by running `generate.py source,inspector` in the `application/feedreader` directory of your qooxdoo SDK or SVN checkout, then open `application/feedreader/inspector/index.html` in your favorite browser.

Now configure the external scripts as described above.

Time to start automating: Click the **Inspect Widget** button in the Inspector's toolbar, then click the Feed Reader's **Add Feed** button. `qx.ui.toolbar.Button[xy]` should now be listed as the inspected widget. If you clicked the button's icon or label, that's fine too.

Click the **plus button** and a new line is added to the test case. Select that line and press **play** and the Add Feed window should open. You might need to move some Inspector windows around to see it.

Now click the **record** button, select **Inspect widget** again and click the upper text field in the Add Feed window. The new command will be added immediately. Select **Inspect Widget** again and click the second text field, then repeat the process for the **Add** button. We're done adding commands, so you can deactivate the **record** button and then close the Add Feed window.

Of course we want to type in the text fields instead of clicking them, so we need to change the commands: Double click the first column of the second row that currently says `qxClick`. Open the dropdown menu that appears and select `qxType`. Now double click this command's `value` cell and enter a title for the new feed to be added, e.g. "Selenium Blogs".

Repeat this step for the next row to define the new feed's URL, e.g. "http://feeds.feedburner.com/Selenium".

That's all the steps we need, so let's watch Selenium work. Set the slider to something around 1.5 seconds, select all four commands in the table and press the **play** button. If all went according to plan, we can click the **export** button to get a Selenese version of our test case to save.

> A debugging tool to inspect a qooxdoo application, featuring an interactive console, an object and widget finder, and a property editor.
>
> Online demo

9.2.4 Simulator (Experimental)

Overview

The purpose of the Simulator component is to help developers rapidly develop and run a suite of simulated user interaction tests for their application with a minimum amount of configuration and using familiar technologies, e.g. qooxdoo-style JavaScript. To do so it uses a combination of qooxdoo's own toolchain, Mozilla's Rhino JavaScript engine and Selenium Remote Control.

Note: The Simulator is a highly experimental feature; the API is by no means finalized. It is included in this qooxdoo release as a preview. Also, the Simulator is *not* intended as a replacement for any existing automated test setup, e.g. using Selenium with JUnit. It is merely one of many ways to run Selenium tests on a qooxdoo application.

Feature Highlights

The Simulator enables developers to:

- Define Selenium test cases by writing qooxdoo classes

- Use the JUnit-style setUp, test*, tearDown pattern
- Define test jobs using the qooxdoo toolchain's configuration system
- Utilize the standard Selenium API and the qooxdoo user extensions to locate and interact with qooxdoo widgets
- Capture and log uncaught exceptions thrown in the tested application
- Use Selenium RC to run tests in many different browser/platform combinations
- Write custom logger classes using qooxdoo's flexible logging system

How it works

Similar to *unit tests*, Simulator test cases are defined as qooxdoo classes living in the application's source directory. As such they support qooxdoo's OO features such as inheritance and nested namespaces. The setUp, testSomething, tearDown pattern is supported, as well as all assertion functions defined by qx.core.MAssert.

The main API that is used to define the test logic is **QxSelenium**, which means the DefaultSelenium API plus the Locator strategies and commands from the qooxdoo user extensions for Selenium.

As with qooxdoo's unit testing framework, the Generator is used to create a test runner application (the Simulator). User-defined test classes are included into this application, which extends qx.application.Native and uses a simplified loader so it can run in Rhino.

A separate Generator job is used to start Rhino and instruct it to load the Simulator application, which uses Selenium's Java API to send test commands to a Selenium RC server (over HTTP, so the server can run on a separate machine). The Server then launches the selected browser, loads the qooxdoo application to be tested and executes the commands specified in the test case.

Setting up the test environment

The following sections describe the steps necessary to set up Simulator tests for an application based on qooxdoo's GUI or inline skeleton.

Required Libraries

The Simulator needs the following external resources to run:

- Java Runtime Environment: Version 1.6 is known to work
- Selenium RC: The required components are selenium-server.jar and selenium-java-client-driver.jar. Versions 1.0 up to and including 2.0a5 are known to work.
- Mozilla Rhino: Versions 1.7R1 and later.
- Qooxdoo User Extensions for Selenium (user-extensions.js).

The Selenium Client Driver (selenium-java-client-driver.jar) and Rhino (js.jar) archives must be located on the same machine as the application to be tested.

The Selenium Server (selenium-server.jar) can optionally run on a physically separate host (see the Selenium RC documentation for details). The qooxdoo user extensions must be located on the same machine as the server.

Note: The qooxdoo User Extensions for Selenium will be moved into the Simulator component for a future release so that it will no longer be necessary to download the file separately.

Generator Configuration

Unlike other framework components, the Simulator isn't ready to run out of the box: The application developer needs to specify the location of the required external libraries (Selenium's Java Client Driver and Mozilla Rhino). This is easily accomplished by redefining the *SIMULATOR_CLASSPATH* macro (in the applicaton's config.json file):

```
"let" :
{
  "SIMULATOR_CLASSPATH" : ["../selenium/selenium-java-client-driver.jar", "../rhino/js.jar"]
}
```

Additional options are available, although their default settings should be fine for most cases. See the *simulate job key reference* for details.

The "settings" section of the "simulation-build" job configures where the AUT is located and how to reach the Selenium RC server that will launch the test browser and run the test commands. The following example shows the minimum configuration needed to build a Simulator application that will test the source version of the current library in Firefox 3 using a Selenium RC server instance running on the same machine (localhost):

```
"simulation-build" :
{
  "settings" :
  {
    "simulator.testBrowser" : "*firefox3",
    "simulator.selServer" : "localhost",
    "simulator.selPort" : 4444,
    "simulator.autHost" : "http://localhost",
    "simulator.autPath" : "/${APPLICATION}/source/index.html"
  }
}
```

See the *job reference* for a listing of all supported settings and their default values.

Note: Since these settings are integrated into the Simulator application by qooxdoo's compile process, the simulation-build job **must** be run again whenever configuration settings were modified. Future versions of the Simulator will get rid of this limitation by using a more flexible configuration approach.

Writing Test Cases

The following articles describe the QxSelenium API in greater detail than can be covered here:

- The qooxdoo user extensions for Selenium
- How to write qooxdoo tests with Selenium

Also, qooxdoo's *Inspector component* can provide assistance to test developers.

Generating the Simulator

The "simulation-build" job explained above is used to generate the Simulator application (in the AUT's root directory):

Note: generate.py simulation-build

Starting the Selenium RC server

The Selenium RC server must be started with the *-userExtensions* command line option pointing to the qooxdoo user extenions for Selenium mentioned above:

```
java -jar selenium-server.jar -userExtensions ../some/path/user-extensions.js
```

Note that the user extension file **must** be named *user-extensions.js*.

Running the Tests

Once the Simulator application is configured and compiled and the Selenium RC server is running, the test suite can be executed using the "simulation-run" job:

```
generate.py simulation-run
```

The Simulator's default logger writes the result of each test to the shell as it's executed. The full output looks something like this:

```
============================================================================
    EXECUTING: SIMULATION-RUN
============================================================================
>>> Initializing cache...
>>> Running Simulation...
>>> Load runtime: 360ms
>>> Simulator run on Thu, 02 Dec 2010 15:57:30 GMT
>>> Application under test: http://localhost/~dwagner/workspace/myApplication/source/index.html
>>> Platform: Linux
>>> User agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.12) Gecko/20101026 Firefox/3.6.12
>>> PASS  myapplication.simulation.DemoSimulation:testButtonPresent
>>> PASS  myapplication.simulation.DemoSimulation:testButtonClick
>>> Main runtime: 11476ms
>>> Finalize runtime: 0ms
>>> Done
```

A framework to develop simulated interaction tests, using Selenium.

CHAPTER

TEN

MIGRATION

10.1 Migration Guide

Migrating from a previous qooxdoo version to a current release often requires nothing more than just running the `migration` job in your application. Yet, some changes between releases may involve manual modifications as detailed in the migration guide of each individual release. The following guide cover both cases.

If you are migrating from a legacy verison of qooxdoo to 1.3.1, namely from a **0.8.2** or prior release, please do a *two-step* migration to 1.3.1. Firstly, migrate to qooxdoo 0.8.3, following the instructions in the corresponding manual. You will need a qooxdoo 0.8.3 SDK to go through the process, so fetch one from the download location. This is necessary as there have been major changes in qooxdoo which require the infrastructure of the intermediate version to bridge. Then, follow the remaining steps in this document.

- **Backup**

  You might want to create a backup of your application files first. The migration process changes source files in place, modifying your code base.

- **Configuration(1)**

  Then, after you have unpacked the new qooxdoo SDK on your system, change references to the framework in your `config.json` and possibly in `generate.py` to point to the new version (look for "QOOXDOO_PATH").

- **Configuration(2)**

  Check the current release notes and those of previous releases between your current version and 1.3.1 for changes to the generator configuration, as they have to be done by hand. Make sure you apply them to your config.json as far as they affect your particular config file. For example, with 0.8.1 the config.json macro `QOOXDOO_PATH` does not include the trailing "framework" part anymore, so make sure to add that. E.g. if you list the qooxdoo framework Manifest.json explicitly in your config using QOOXDOO_PATH, make sure "/framework" is appended after the macro reference.

  - Alternatively, particularly if you config.json is rather small, create a *separate gui skeleton* elsewhere and copy its config.json over to your application, and port the config settings from your old configuration to this file. This might be the simpler approach.

- **Run Migration**

  Then change to your application's top-level directory and invoke the command

  ```
  generate.py migration
  ```

- Follow the instructions of the migration script, particularly allow the cache to be deleted. For more information about this script, see the *corresponding job description*.

- **Migration Log**

  Check the `migration.log` which is created during the run of the migration script. Check all hints and deprecation warnings in the log and apply them to your code.

You now have an up-to-date source tree in your application. Run

```
generate.py source
```

to check that the generation process goes through and test your application in the browser.

CHAPTER

ELEVEN

REFERENCES

11.1 Core

11.1.1 Class Declaration Quick Ref

This is a quick reference for the various features of a qooxdoo class declaration. It uses an EBNF-like syntax.

Properties, a particular part of the class declaration, have quite an extensive sub-spec, and are therefore factored out to their *own page*.

```
class_decl      := 'qx.Class.define' '(' '"' <name.space.ClassName> '"' ','
                     '{' { feature_spec ',' } '}'
                   ')'

feature_spec    :=
                   'type'        ':' type_spec        |
                   'extend'      ':' extend_spec      |
                   'implement'   ':' implement_spec   |
                   'include'     ':' include_spec     |
                   'construct'   ':' construct_spec   |
                   'statics'     ':' statics_spec     |
                   'properties'  ':' properties_spec  |
                   'members'     ':' members_spec     |
                   'settings'    ':' settings_spec    |
                   'variants'    ':' variants_spec    |
                   'events'      ':' events_spec      |
                   'defer'       ':' defer_spec       |
                   'destruct'    ':' destruct_spec

type_spec       := 'static' | 'abstract' | 'singleton'

extend_spec     := <name.of.SuperClass>

implement_spec  := <name.of.Interface> |
                   '[' <name.of.Interface1> ',' <name.of.Interface2> ','
                       ... ']'

include_spec    := <name.of.Mixin> |
                   '[' <name.of.Mixin1> ',' <name.of.Mixin2> ',' ... ']'

construct_spec  := js_function_value

statics_spec    := c_map
```

```
properties_spec := ? see separate properties quick ref ?

members_spec    := c_map

settings_spec   := '{' { '"' <settings_name> '"' ':'
                         ( js_primitive_value | js_reference_value )
                         ',' } '}'

variants_spec   := 'qx.Variant.select' '(' '"' <variantName> '"' ','
                     '{' { '"' <variantvalue_spec> '"' ':' <selectValue>
                           ',' } '}'
                   ')'

events_spec     := '{' { '"' <event_name> '"' ':' '"' qx_event_type '"'
                       ',' } '}'

defer_spec      := js_function_value

destruct_spec   := '{'
                    [ 'this._disposeFields'     '(' <fields_list> ')' ';' ]
                    [ 'this._disposeObjects'    '(' <fields_list> ')' ';' ]
                    [ 'this._disposeObjectDeep' '(' <deep_field> ')'      ]
                    '}'

c_map           := '{' { <key> ':' [ js_primitive_value |
                                     js_reference_value |
                                     js_function_value  |
                                     variants_spec      ] ',' } '}'

js_function_value   := ? Javascript anonymous function 'function (...) {...}' ?
js_primitive_value  := ? any value from the Javascript primitive types ?
js_reference_value  := ? any value from the Javascript reference types ?
qx_event_type       := ? any qooxdoo event type class name, e.g.
                         'qx.event.type.DataEvent' ?
```

11.1.2 Interfaces Quick Ref

This is a quick reference for the various features of a qooxdoo interface declaration. It uses an EBNF-like syntax.

It is much like a class declaration, with a more limited set of features. Properties are just names with empty map values.

```
interface_decl  := 'qx.Interface.define' '(' '"' <name.space.InterfaceName> '"' ','
                     { feature_spec ',' }
                   ')'

feature_spec    := 'extend'     ':' extend_spec     |
                   'statics'    ':' statics_spec    |
                   'properties' ':' properties_spec |
                   'members'    ':' members_spec    |
                   'events'     ':' events_spec

extend_spec     := <name.of.SuperInterface> |
                   '[' <name.of.SuperInterface1> ',' <name.of.SuperInterface2>
                       ',' ... ']'

statics_spec    := '{' { '"' <upper_case_key> '"' ':' js_primitive_value ',' } '}'
```

```
properties_spec := '{' { '"' <property_name> '"' ':' '{}' ',' } '}'

members_spec    := c_map

events_spec     := '{' { '"' <event_name> '"' ':' '"' qx_event_type '"'
                        ',' } '}'

c_map           := '{' { <key> ':' [ js_primitive_value |
                                     js_reference_value |
                                     js_function_value  |
                                     variant_spec       ] ',' } '}'

js_function_value   := ? Javascript anonymous function 'function (...) {...}' ?
js_primitive_value  := ? any value from the Javascript primitive types ?
js_reference_value  := ? any value from the Javascript reference types ?
qx_event_type       := ? any qooxdoo event type class name, e.g.
                         'qx.event.type.DataEvent' ?
```

11.1.3 Mixin Quick Ref

This is a quick reference for the various features of a qooxdoo mixin declaration. It uses an EBNF-like syntax.

It is much like a class declaration, with a more limited set of features. Properties are documented on their *own page*.

```
mixin_decl      := 'qx.Mixin.define' '(' '"' <name.space.MixinName> '"' ','
                     { feature_spec ',' }
                   ')'

feature_spec    :=
                   'include'    ':' include_spec     |
                   'construct'  ':' construct_spec   |
                   'statics'    ':' statics_spec     |
                   'properties' ':' properties_spec  |
                   'members'    ':' members_spec     |
                   'events'     ':' events_spec      |
                   'destruct'   ':' destruct_spec

include_spec    := <name.of.Mixin> |
                   '[' <name.of.Mixin1> ',' <name.of.Mixin2> ',' ... ']'

construct_spec  := js_function_value

statics_spec    := c_map

properties_spec := ? see separate properties quick ref ?

members_spec    := c_map

events_spec     := '{' { '"' <event_name> '"' ':' '"' qx_event_type '"'
                        ',' } '}'

destruct_spec   := '{'
                    [ 'this._disposeFields'     '(' <fields_list> ')' ';' ]
                    [ 'this._disposeObjects'    '(' <fields_list> ')' ';' ]
                    [ 'this._disposeObjectDeep' '(' <deep_field> ')'      ]
                    '}'
```

```
c_map            := '{' { <key> ':' [ js_primitive_value |
                                      js_reference_value |
                                      js_function_value  |
                                      variants_spec      ] ',' } '}'

js_function_value   := ? Javascript anonymous function 'function (...) {...}' ?
js_primitive_value  := ? any value from the Javascript primitive types ?
js_reference_value  := ? any value from the Javascript reference types ?
qx_event_type       := ? any qooxdoo event type class name, e.g.
                         'qx.event.type.DataEvent' ?
```

11.1.4 Properties Quick Reference

This is a quick reference for the various property features available in qooxdoo.

Properties are declared in the constructor map of the class as a dedicated key-value pair (here called `properties_decl`). This is the quick reference for properties_decl (expressed in an EBNF'ish way):

```
properties_decl  := 'properties' ':' properites_map

properties_map   := '{' { prop_spec ',' } '}'
prop_spec        := '"' <property_name> '"' ':' '{'
                      { property_feature ',' } '}'

property_feature := nullable_spec      |
                    apply_spec         |
                    event_spec         |
                    init_spec          |
                    refine_spec        |
                    check_spec         |
                    themeable_spec     |
                    inheritable_spec   |
                    group_spec         |
                    mode_spec          |
                    validate_spec      |
                    dereference_spec

nullable_spec    := 'nullable'     ':' bool_val
apply_spec       := 'apply'        ':' '"' <FunctionName> '"'
event_spec       := 'event'        ':' '"' <EventName> '"'
init_spec        := 'init'         ':' <InitVal>
refine_spec      := 'refine'       ':' bool_val

check_spec       := 'check'        ':' '"' type_spec '"'        |
                                       '"' <ClassName> '"'      |
                                       '"' <InterfaceName> '"'  |
                                       enum_spec                |
                                       inline_function          |
                                       '"' bool_expression '"'

validate_spec    := 'validate'     ':' '"' <FunctionName> '"'   |
                                       '<Function>'

dereference_spec := 'dereference'     ':' bool_val

themeable_spec   := 'themeable'    ':' bool_val
inheritable_spec := 'inheritable'  ':' bool_val
```

```
group_spec        := 'group'        ':' enum_spec
mode_spec         := 'mode'         ':' '"' 'shorthand' '"'

type_spec         := 'Boolean' | 'String' | 'Number' | 'Integer' | 'Float' |
                     'Double'  | 'Object'| 'Array'   | 'Map'     | 'Class' |
                     'Mixin'   | 'Interface'         | 'Theme'   | 'Error' |
                     'RegExp'  | 'Function'          | 'Date'    | 'Node'  |
                     'Element' | 'Document'          | 'Window'  | 'Event'

bool_val          := 'true' | 'false'
enum_spec         := '[' <val1>',' <val2> ',' ... ',' <valN> ']'
inline_function   := ? JavaScript anonymous function 'function (..) { ... }' ?
bool_expression   := ? JavaScript expression evaluating to true/false ?
```

11.1.5 Array Reference

qooxdoo has a few classes that concern arrays. Some of them are special wrappers and others are extensions. Here is a list of all classes which have something to do with arrays in qooxdoo.

Data binding specific array

- qx.data.Array: The data array is a special array used in the data binding context of qooxdoo. It does not extend the native array of JavaScript but is a wrapper for it. All the native methods are included in the implementation and it also fires events if the content or the length of the array changes in any way. Also the *.length* property is available on the array.

Extension of the native array

- qx.type.BaseArray: This class is the common superclass for all array classes in qooxdoo. It supports all of the shiny 1.6 JavaScript array features like *forEach* and *map*. This class may be instantiated instead of the native Array if one wants to work with a feature-unified Array instead of the native one. This class uses native features wherever possible but fills all missing implementations with custom code.
- qx.type.Array: An extended array class which adds a lot of often used convenience methods to the regular array like *remove* or *contains*.

Utility methods

- qx.lang.Array: Provides static helper functions for arrays with a lot of often used convenience methods like *remove* or *contains*.

Extending the native array's prototype

- qx.lang.Core: Adds some methods to array to lift every browser to the same level. The methods are:
  - indexOf
  - lastIndexOf
  - forEach
  - filter
  - map

  - some
  - every
- qx.lang.Generics : Support string/array generics as introduced with JavaScript 1.6 for all browsers.
  - join
  - reverse
  - sort
  - push
  - pop
  - shift
  - unshift
  - splice
  - concat
  - slice
  - indexOf
  - lastIndexOf
  - forEach
  - map
  - filter
  - some
  - every

11.1.6 Framework Generator Jobs

This page describes the jobs that are available in the framework. Mainly this is just a reference list with short descriptions of what the jobs do. To find out more about predefined jobs for custom applications, see the *Default Generator Jobs*.

Framework Jobs

These jobs can be invoked in the *framework/* directory with the generator, as `generate.py <jobname>`.

api

Create api doc for the framework.

api-data

Create the api data for the framework. Can take individual class names as further arguments.

clean

Remove local cache and generated .js files.

distclean

Remove the cache and all generated artefacts of the framework (api, test, ...).

fix

Normalize whitespace in .js files of the framework (tabs, eol, ...).

images

Run the image clipping and combining of framework images.

lint

Check the source code of the frameworks .js files (except the tests).

lint-test

Check the source code of the test .js files of the framework.

qxoo-build

Creates a single file containing all the qooxdoo classes of the OO layer. This file can be used in non-browser environments.

qxoo-noopt

A non-optimized version of *qxoo-build*, for debugging.

test

Create a test runner app for unit tests of the framework.

test-source

Create a test runner app for unit tests (source version) of the framework.

test-inline

Create an inline test runner app for unit tests of the framewrok.

translation

Create the .po files of the framework.

11.2 GUI Toolkit

11.2.1 Widget Reference

Core Widgets

Widget, *Spacer*, *ScrollBar*

Content Widgets

Label, *Image*, *Atom*, *Tree*, *Table*

Container Widgets

Composite, *Scroll*, *Stack*, *SlideBar*, *Resizer*

Building Blocks

Toolbar, *TabView*, *SplitPane*, *GroupBox*, *MenuBar*

Popups

PopUp, *ToolTip*, *Menu*, *Window*

Embed Widgets

Canvas, *HTML Embed*, *Iframe*

Form Widgets

Button, *ToggleButton*, *RepeatButton*, *HoverButton*, *SplitButton*, *MenuButton*

TextField, *PasswordField*, *Spinner*, *DateField*, *TextArea*

ComboBox, *SelectBox*

CheckBox, *List*, *Slider*

Virtual Widgets

Virtual List

Indicators

ProgressBar

Widget Reference List

Atom The Atom groups an image with a label with support for different alignments. It is a building block for many other widgets like Buttons or Tooltips.

Preview Image

Features

- Configurable spacing between icon and label
- Toggle display of "image" "label" or "both"
- Configurable icon position

Demos Here are some links that demonstrate the usage of the widget:

- A simple Atom demo

API Here is a link to the API of the Widget: qx.ui.basic.Atom

Button A Button widget is used to display plain text and/or an icon. The button supports mouse and key events.

Preview Image

Features

- Contain text and/or icon.
- Mouse and keyboard support.
- Ellipsis: If the label does not fit into the widget bounds an ellipsis ("...") is rendered at the end of the label.

Description The button widget is a normal widget for a GUI. The button supports plain text and icon. Also it is possible to handle user interactions with mouse and keyboard.

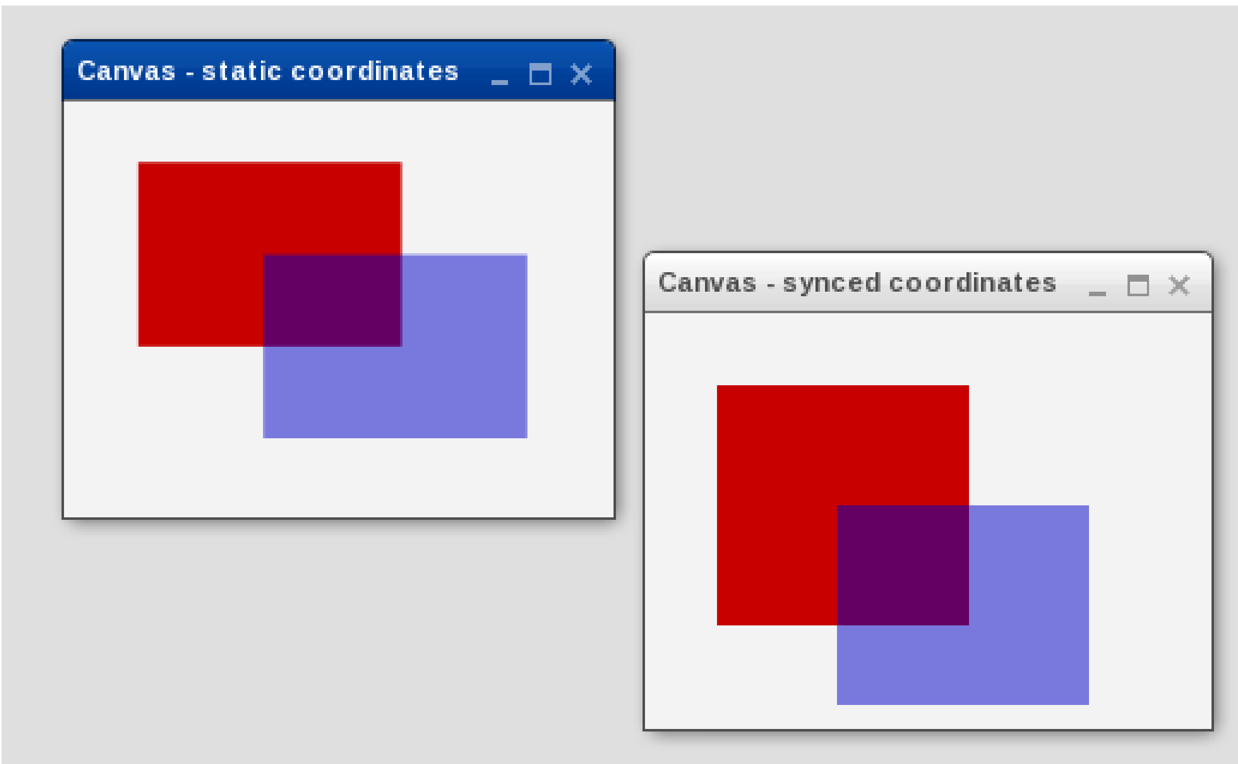
Demos Here are some links that demonstrate the usage of the widget:

- A button demo with differently configured buttons
- A window demo which using button

API Here is a link to the API of the Widget:
qx.ui.form.Button

Canvas This widget embed the HTML canvas element.

Note: It does not work with Internet Explorer



Preview Image

Features Since this widget is embedding the HTML canvas element the core features of this widget are limited by the canvas element itself respective by the implementation of the different browsers. However, the widget offers these features on top:

- fires a `redraw` event whenever the dimensions of the canvas element has changed or the canvas element needs an update
- update method for the canvas element
- width and height of the canvas element as properties
- support for synronized widht and height coordinates

Description Taken from the WHATWG website: "The canvas element represents a resolution-dependent bitmap canvas, which can be used for rendering graphs, game graphics, or other visual images on the fly."

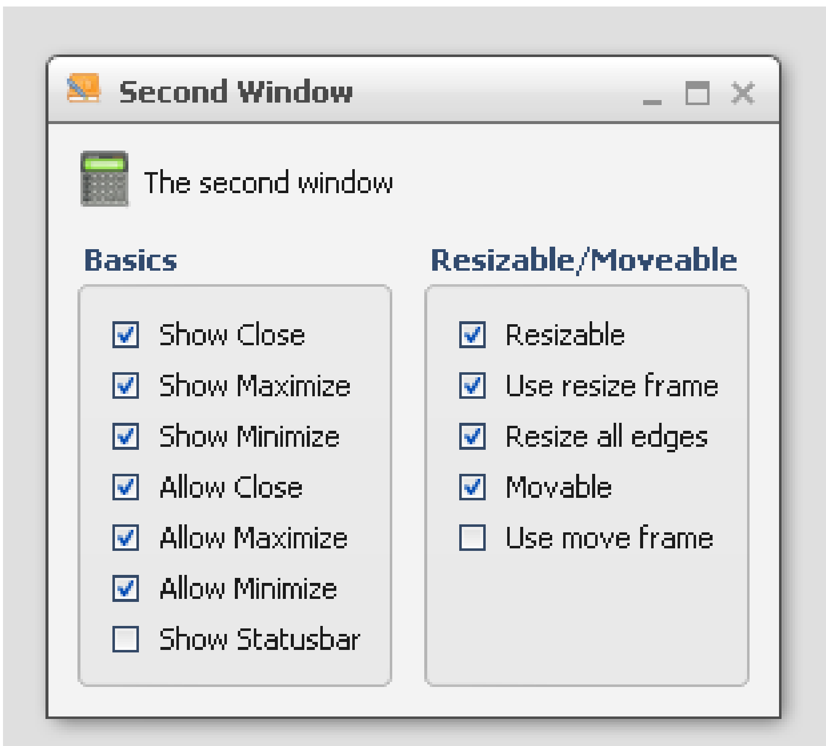
Demos Here are some links that demonstrate the usage of the widget:

- Canvas demo

API Here is a link to the API of the Widget:
Canvas API

CheckBox A CheckBox widget for Boolean values.

Preview Image



Features

- Mouse and keyboard control.
- Ellipsis: If the label does not fit into the widget bounds an ellipsis ("...") is rendered at the end of the label.

Description The CheckBox is a common widget found in many GUI applications. A CheckBox can be checked or not checked, either by mouse or keyboard. When the tri-state mode is enabled, there is an additional third state. The third state means that the CheckBox was neither checked nor unchecked, i.e. the state of the CheckBox is undetermined.

The CheckBox supports an optional plain text.

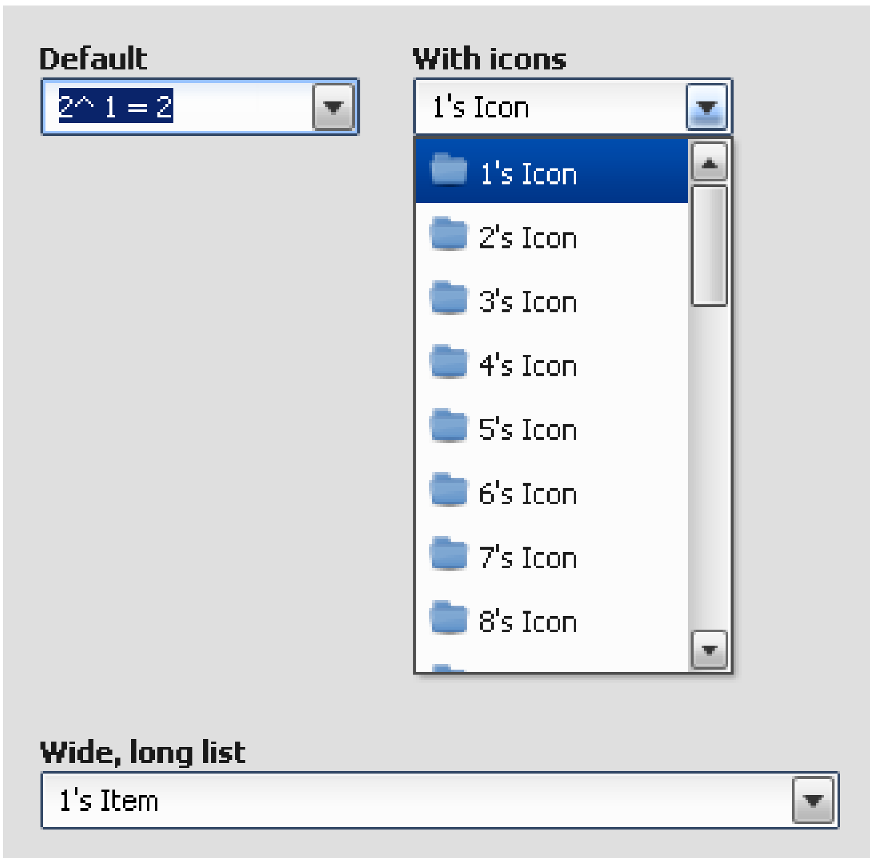
Also it is possible to combine a CheckBox with a TreeItem to construct a complex widget.

Demos Here are some links that demonstrate the usage of the widget:

- CheckBoxes used in a GroupBox
- CheckBoxes used in a GroupBox to control a window
- A small dialog Demo
- ComboBox combined with a TreeItem

API Here is a link to the API of the Widget:
qx.ui.form.CheckBox

ComboBox A ComboBox widget is used to select items from a list or allow customer input. The items in a ComboBox supports plain text and/or icons.



Preview Image

Features

- Mouse and keyboard support.
- Items with plain text and/or icons
- Ellipsis: If the label does not fit into the widget bounds an ellipsis (”...”) is rendered at the end of the label.

Description A ComboBox is like a *TextField* with a drop down of predefined values. The main difference to the *SelectBox* is that the user can enter individual values or choose from the predefined ones. The items in the predefined list supports plain text and/or icons. The items which can be added to the list are `qx.ui.form.ListItem` items.

Please note that the ComboBox supports no auto- completion.

Demos Here are some links that demonstrate the usage of the widget:

- ComboBox demo
- Form demo

API Here is a link to the API of the Widget:
qx.ui.form.ComboBox

Composite The Composite is a generic container widget. It exposes all methods to set layouts and to manage child widgets as public methods. Composites must be configured with a layout manager to define the way the widget's children are positioned.

Features

- Public methods to manage child widgets (add, remove, ...)
- Public `setLayout` method to define the Composite's layout manager

Description Composites are used to manually compose widgets. They are always used in combination with a layout manager. The general behavior of this widget is controlled by this layout manager.

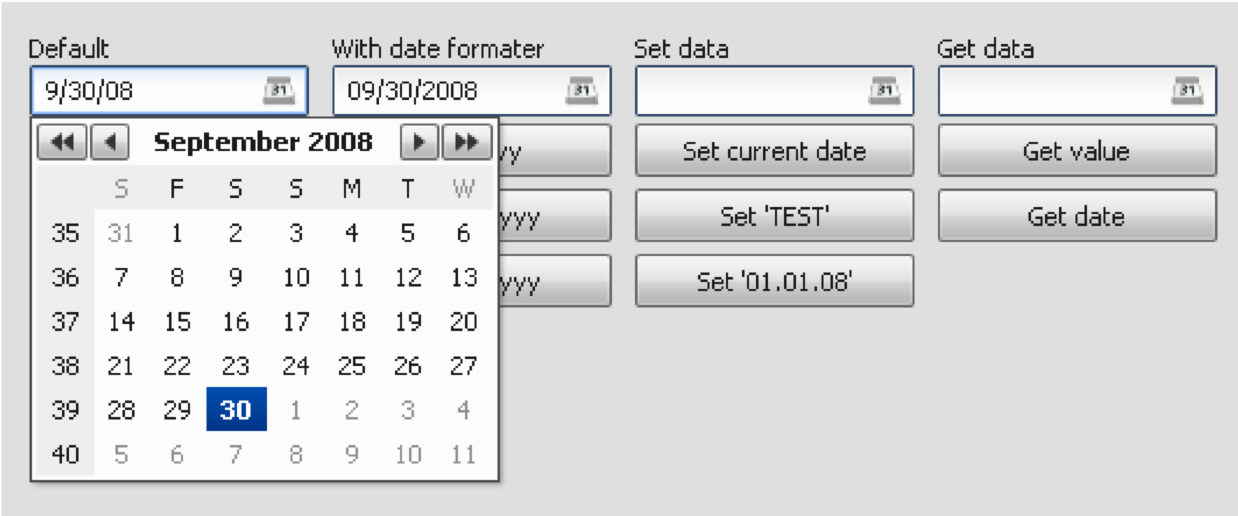
Demos Any of the layout demos use Composites:

Here are some links that demonstrate the usage of the widget:

- The first layout demo. Any other layout demo uses Composites as well.

API Here is a link to the API of the Widget:
qx.ui.container.Composite

DateField A DateField widget can be used for date input. The input can be done in to kinds. The first kind is to chose a date from a date chose, which is a part of the DateField. The second kind is to write the date direct in the input field. .. _pages/widget/datefield#preview_image:



Preview Image

Features

- Mouse and keyboard support.
- Own date format.

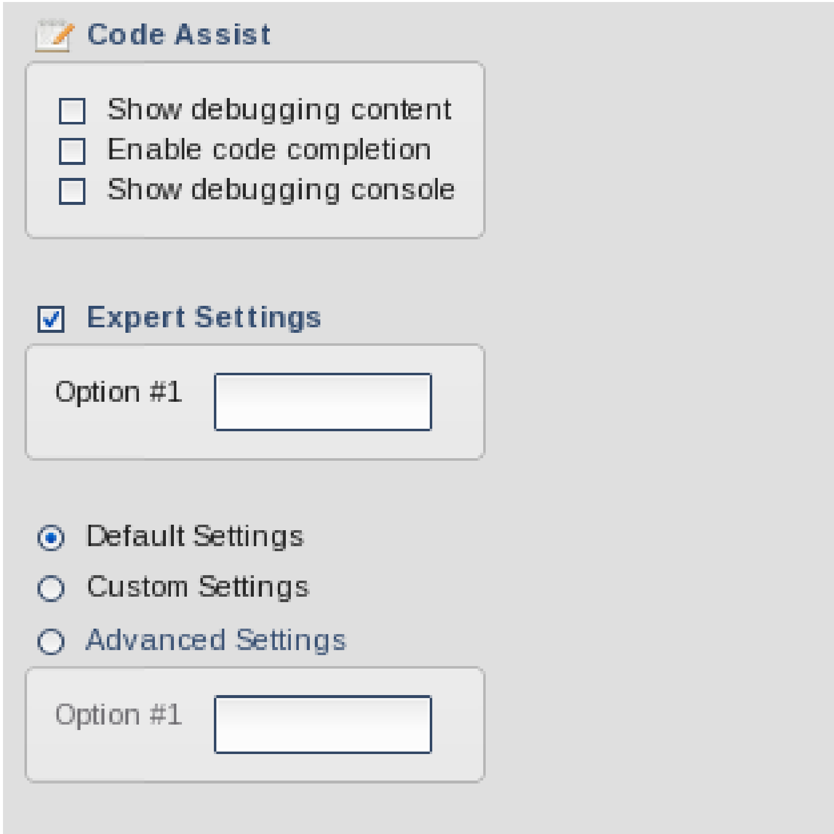
Description A DateField has a `qx.util.format.DateFormat` which is used to format the date to a string. The formatted string is show in the input field. The input can be edit directly in the input filed or selecting a date with the data chooser. The date chooser can be pop up by clicking the calender icon.

Demos Here are some links that demonstrate the usage of the widget:

- DateField Demo
- Form demo

API Here is a link to the API of the Widget:
qx.ui.form.DateField

GroupBox A Groupbox is a widget to group a set of form elements in a visual way.



Preview Image

Features

- Different legend types
- icon and text
- additional check boxes
- additional radio buttons

Description The GroupBox offers the possibility to visual group several form elements together. With the use of a legend which supports both text and icon it is easy label the several group boxes to give the user a short description of the form elements.

Additionally it is possible to use checkboxes or radio-buttons within the legend to enable or disable the connected groupBox (and their child elements) completely. This feature is most important for complex forms with multiple choices.

Demos Here are some links that demonstrate the usage of the widget:

- Demo showing all groupBox types

API Here is a link to the API of the Widget:
qx.ui.groupbox.GroupBox

HoverButton The HoverButton is an *Atom*, which fires repeatedly execute events while the mouse is over the widget.

Preview Image

Features

- Contain text and/or icon.
- Ellipsis: If the label does not fit into the widget bounds an ellipsis ("...") is rendered at the end of the label.
- Event interval is adjustable.

Description The HoverButton is an *Atom*, which fires repeatedly execute events while the mouse is over the widget. The interval time for the HoverButton event can be configured by the developer.

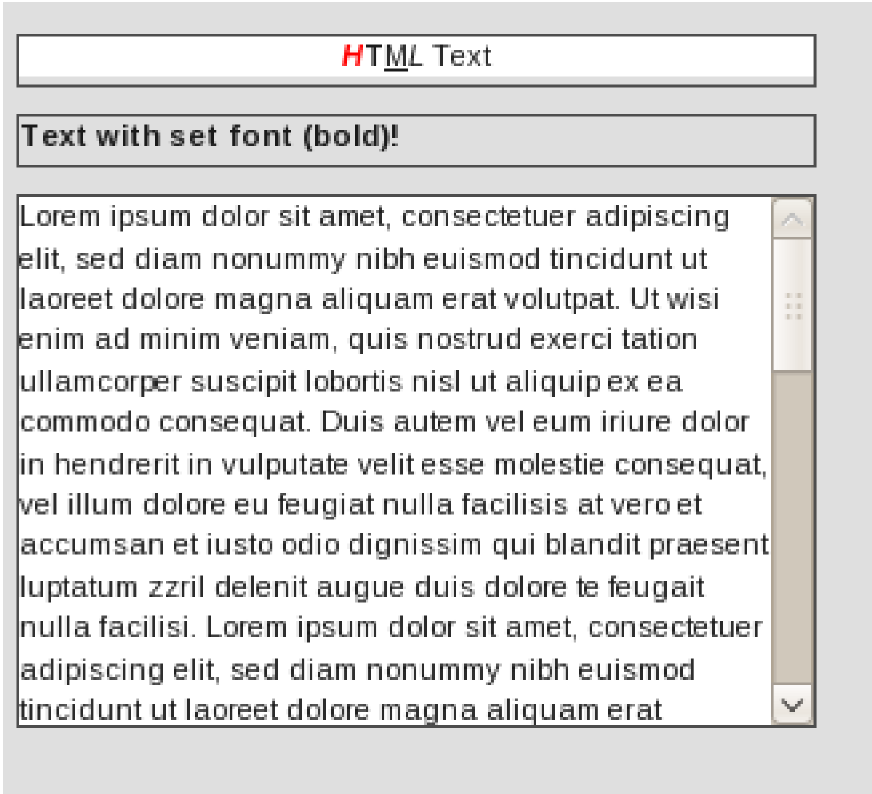
Demos Here are some links that demonstrate the usage of the widget:

- Button demo with all supported buttons
- Form showcase demo

API Here is a link to the API of the Widget:
qx.ui.form.HoverButton

HTML Embed The Html widget embeds plain HTML code into the application.

Preview Image



Features

- displays any valid HTML code
- CSS class support
- control whether the content is focusable
- control whether the content is selectable
- overflow support
- data event `changeHtml` is dispatched whenever content changes

Description The HTML embed can display any valid HTML code and implements some useful features like focus- and selection-control on top of it.

If you want to display a large amount of HTML code you can additionally use the overflow control to prevent the widget from eating up too much space within your application. This makes the seamless integration as easy as possible.

If you want to manipulate the styling of the displayed HTML code you can easily set a CSS class name to have the full control of the HTML.

Demos Here are some links that demonstrate the usage of the widget:

- HTML embed demo

API Here is a link to the API of the Widget:
HTML Embed API

Iframe Container widget for internal frames (iframes). An iframe can display any HTML page inside the widget.

Preview Image

Features

- can display any HTML page
- fires a `load` event when the page fully loaded
- integrates a blocker element to prevents the iframe to handle key or mouse events

Description The iframe is a container widget for displaying any HTML page. It integrates seamlessly in your application though it can be styled like any other qooxdoo widget and offers an `load` event to control the page that's loaded within the widget. And the built-in blocker element prevents the native iframe element to handle any key or mouse event to ensure that e.g. the user navigates away by clicking a hyperlink.

Demos Here are some links that demonstrate the usage of the widget:

- Iframe demo

API Here is a link to the API of the Widget:
API for Iframe

Image As the name suggest, the Image widget is used to display image files.

Preview Image 

Features

- Scaling the image
- Image clipping (combine multiple images into one single image)
- Auto sizing
- Configurable second image for the disabled state
- Support for PNG files with alpha transparency in all browsers (including IE6)

Demos Here are some links that demonstrate the usage of the widget:

- Image demo

API Here is a link to the API of the Widget:
qx.ui.basic.Atom

Label The Label widget is used to display either plain text or rich text with HTML markup.

Preview Image 

Features

- Auto sizing
- Ellipsis: If the label does not fit into the widget bounds an ellipsis ("...") is rendered at the end of the label. (Only in text mode)
- "height for width": If the widget's width is too small to display the text in one line the text is wrapped and a new size hint is calculated. (Only in HTML mode)
- Configurable fonts, text colors and text alignment

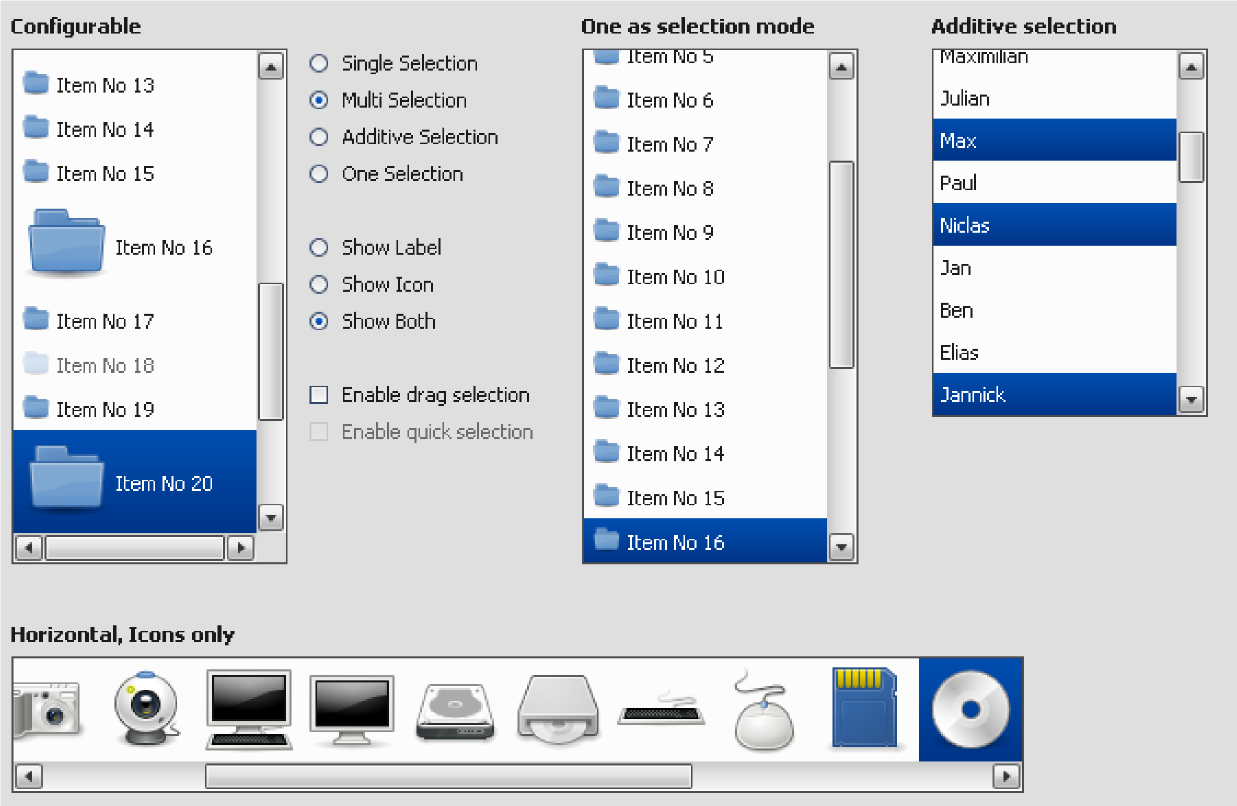
Description The Label supports two different modes. The text and the HTML mode. The mode can be set by using the `rich` property. Which mode to use depends on the required features. If possible the text mode should be preferred because in this mode the text size calculation is faster.

Demos Here are some links that demonstrate the usage of the widget:

- A label demo with differently configured labels
- Height for width demo
- Label reflow

API Here is a link to the API of the Widget:
qx.ui.basic.Label

List A List widget has items with plain text and/or icon.



Preview Image

Features

- Horizontal and vertical orientation.
- Single selection.
- Multi selection.
- Additive selection.
- One selection.
- Drag selection.
- Quick selection.
- Items with plain text and/or icon.
- Context menu support.

Description A List widget can be used to show a list of items. These items could selected in different modes:

- `single`: Only one or none could be selected.
- `multi`: One, more or none could be selected.
- `additive`: The same selection like `multi`, but each item, which the user clicked on it is added or removed to the selection.

- one: The same selection like single, but one must selected.

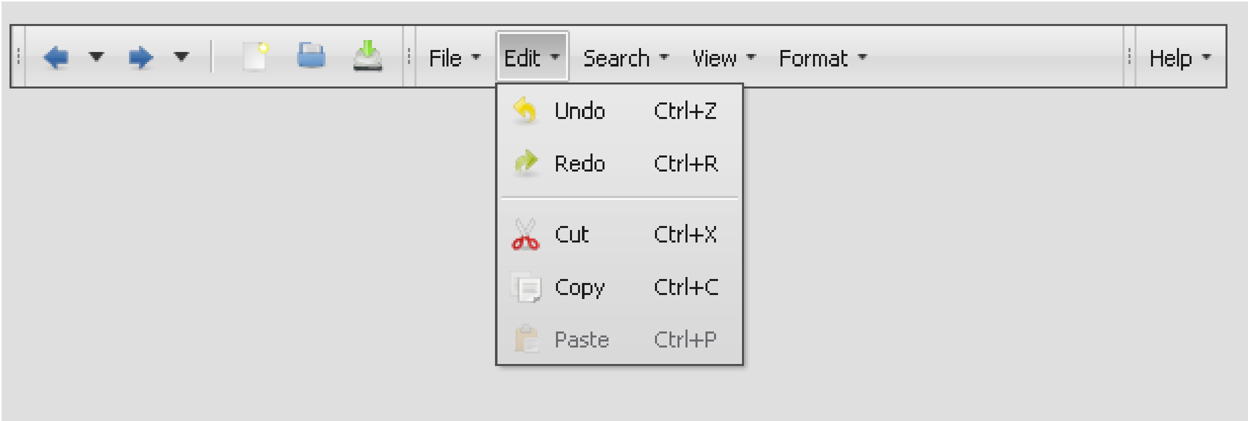
The item which are added to the list are ListItem. For more details see: ListItem.

Demos Here are some links that demonstrate the usage of the widget:

- List Demo
- Lists with Drag and Drop
- List with re-size support

API Here is a link to the API of the Widget:
qx.ui.form.List

Menu The Menu is a widget that contains different widgets to create a classic menu structure. The menu is used from different widget, that needs a menu structure e.q. MenuBar.



Preview Image

Features

- On demand scrolling if the menu doesn't fit on the screen
- Menu items with text and/or icon.
- Each menu item can have a command for keyboard support.
- Menu items can have submenus.

The menu can contain different item types:

- Normal buttons
- CheckBox buttons
- RadioButtons
- Separators

Description The Menu widget is used in combination with other widgets. The other widgets has an instance from the menu and it's shown by user interactions. Each item in a menu can get an command key, that is used to get keyboard support for the user.

Here a some widgets that use a menu for user interaction:

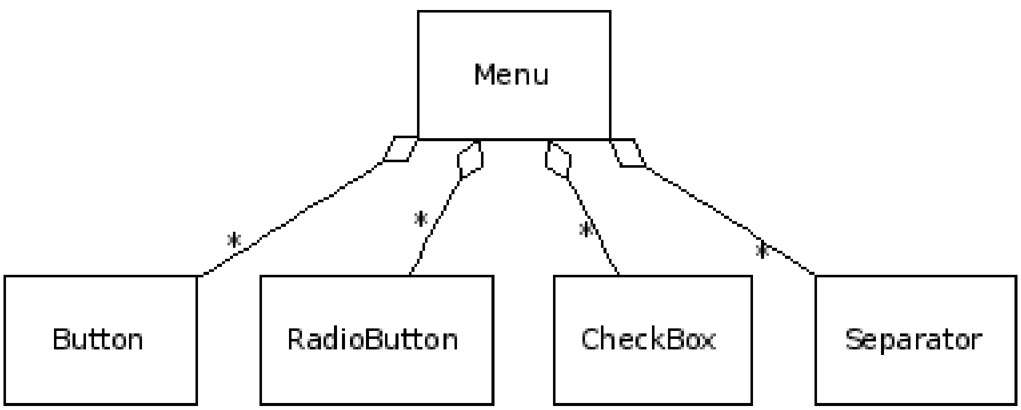
- *MenuBar*
- *Toolbar*
- *MenuButton*
- *SplitButton*
- *List*

The package `qx.ui.menu` has a collection of needed classes for creating a menu structure. The `qx.ui.menu.Menu` class is the container class for the menu structure and has items as child. Here are some item that can be used to create the structure:

- Button
- CheckBox
- RadioButton
- Separator

To create a submenu structure, each item (but not separator) can contain a menu to realize the submenu structure.

Diagram

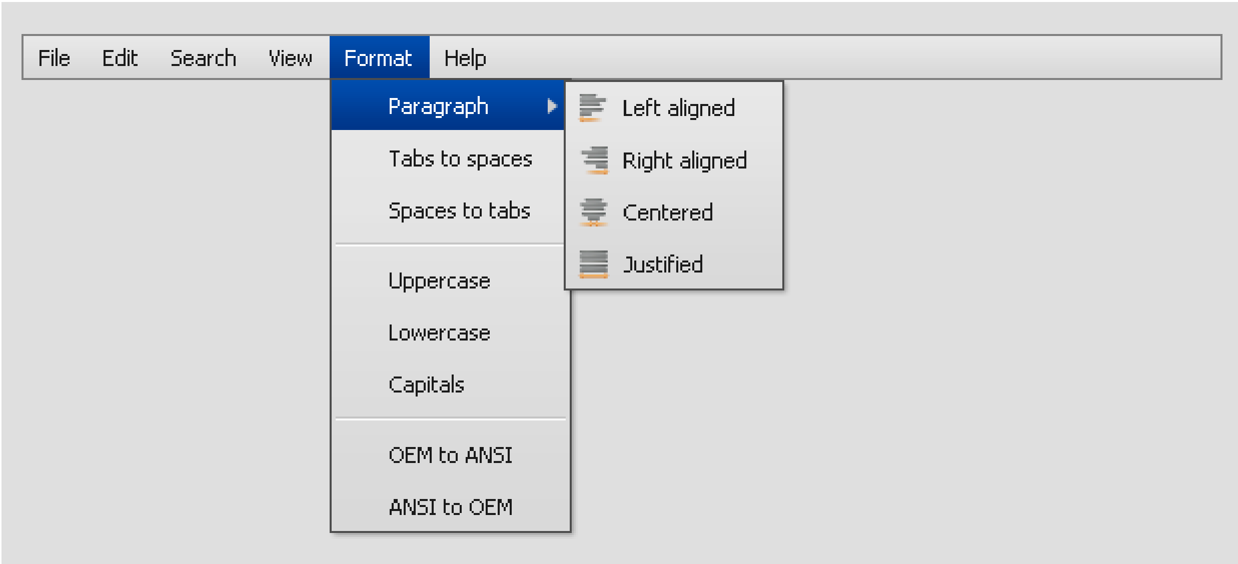


Demos Here are some links that demonstrate the usage of the widget:

- Some different widgets that use the menu functionality
- Menus used in a MenuBar

API Here is a link to the API of the Widget:
qx.ui.menu.Menu

MenuBar The MenuBar is a Widget to create a classic menu bar for an application.



Preview Image

Features

- Buttons as menu items with label and/or icon.

Description The MenuBar contains items `qx.ui.menubar.Button` to open a submenu `qx.ui.menu.Menu` that can handle user interactions. For more information about menus see *Menu*.

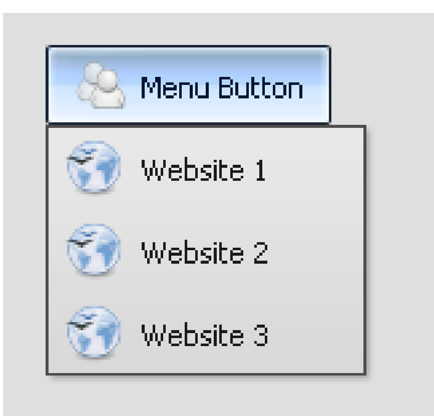
Demos Here are some links that demonstrate the usage of the widget:

- MenuBar with all features

API Here is a link to the API of the Widget:
qx.ui.menubar.MenuBar

MenuButton The MenuButton looks like a normal button, but it opens a menu when clicking on it.

Preview Image



Features

- Contain text and/or icon.
- Mouse support.
- Ellipsis: If the label does not fit into the widget bounds an ellipsis ("...") is rendered at the end of the label.
- Menu support.

Description The MenuButton looks like a normal button, but it opens a menu when clicking on it.

For using a menu see: *Menu*

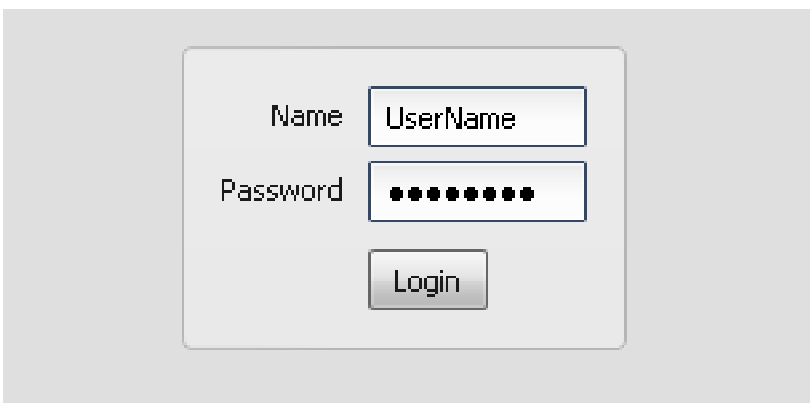
Demos Here are some links that demonstrate the usage of the widget:

- Menu demo that contains a MenuButton
- Form demo

API Here is a link to the API of the Widget:
qx.ui.form.MenuButton

PasswordField The PasswordField widget is a special TextField witch show the input hidden.

Preview Image



Features

- Hide password
- Mouse and keyboard control.
- Set maximum input length.
- Read only support.

Description The PasswordField is a special TextField for password input. The PasswordField hide the text input.

The act is the same like the TextField, for more details see: *TextField*

Demos Here are some links that demonstrate the usage of the widget:

- Login dialog
- Show a form demo

API Here is a link to the API of the Widget:
qx.ui.form.PasswordField

PopUp Popups are widgets, which can be placed on top of the application.

Preview Image 

Features

- Auto hide property

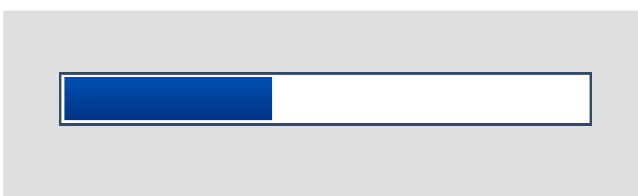
Description Popups are automatically added to the application root and are used to display menus, the lists of combo or select boxes, tooltips, etc.

Demos Here are some links that demonstrate the usage of the widget:

- Simple example for the PopUp widget

API Here is a link to the API of the Widget:
qx.ui.popup

ProgressBar The progress bar is an indicator widget.

Preview Image 

Description The Progress bar is designed to simply display the current % complete for a process. It fires 2 events. When the % changes or when the process is complete.

The Value is limited between 0 and Maximum value. It's not allowed to set a Maximum value of 0. If you set a Maximum value bigger than 0, but smaller than Value, it will be limited to Value.

Here's an example. We create a default progress bar (value is 0, and the maximum value is 100). We then listen to change event and complete event. The change event is fired every time the % complete is changed, so we can see the new value. If the process is 100% complete the complete event is fired.

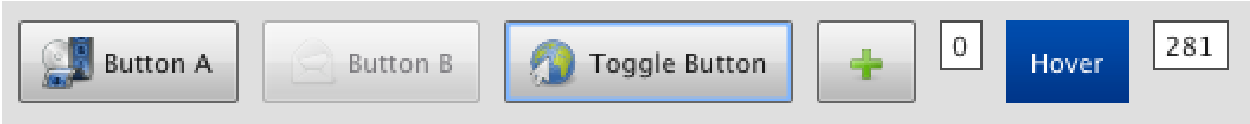
```
var pb = new qx.ui.indicator.ProgressBar();
this.getRoot().add(pb, { left : 20, top: 20});

pb.addListener("change", function(e) {
  this.debug(e.getData()); // % complete
  this.debug(pb.getValue()); // absolute value
});

pb.addListener("complete", function(e) {
  this.debug("complete");
});

//set a value
pb.setValue(20);
```

Demos Here are some links that demonstrate the usage of the widget:

- Simple example for the ProgressBar widget

API Here is a link to the API of the widget:
qx.ui.indicator.ProgressBar

RepeatButton The RepeatButton is a special button, which fires an event, while the mouse button is pressed on the button.



Preview Image

Features

- Contain text and/or icon.
- Mouse support.
- Ellipsis: If the label does not fit into the widget bounds an ellipsis ("...") is rendered at the end of the label.
- Event interval is adjustable.

Description The RepeatButton is a special button, which fires an event, while the mouse button is pressed on the button. The interval time for the RepeatButton event can be configured by the developer.

Demos Here are some links that demonstrate the usage of the widget:

- Button demo with all supported buttons
- Form showcase demo

API Here is a link to the API of the Widget:
qx.ui.form.RepeatButton

Resizer The Resizer is a resizable container widget.

Preview Image 

Features

- Configurable whether all four edges are resizable or only the right and bottom edge
- Live resize or resizing using a resize frame (like in the screen shot)
- Sensitivity configurable i.e. The number of pixels on each side of a resize edge, where the resize cursor is shown.

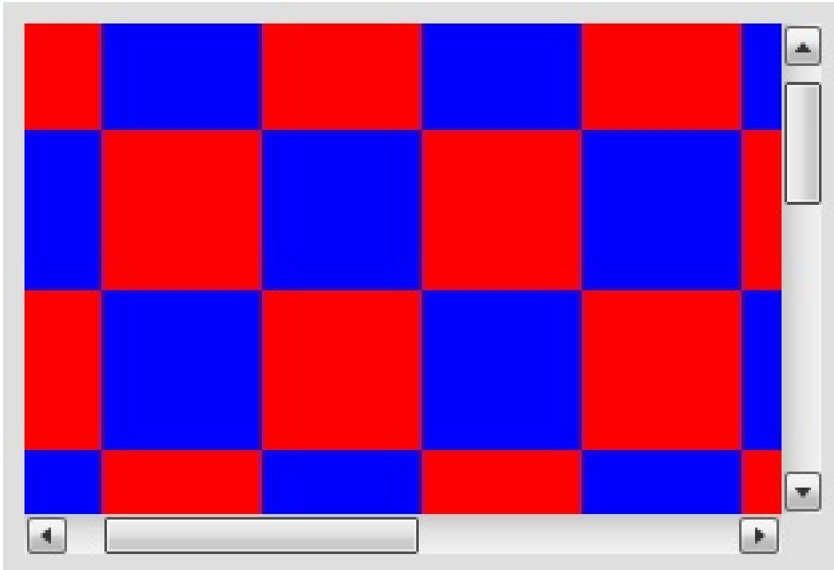
Description The Resizer is a generic container just like a *Composite*, which can be resized by using the mouse. Either all edges or only the right and bottom edge can be configured to be resizable.

Demos Here are some links that demonstrate the usage of the widget:

- Resizer demo

API Here is a link to the API of the Widget:
qx.ui.container.Resizer

Scroll `Scroll` is a container, which allows vertical and horizontal scrolling if the content is larger than the container.

Preview Image 

Features

- Themeable scroll bars
- Scroll bar visibility can be set independently for the X- and Y-axis. Possible values are `auto` (default), `on` and `off`

Description This widget can be used if the container's content is larger than the container itself. In this case vertical or horizontal scroll bars are displayed as needed.

Note that this class can only have one child widget and no configurable layout. The layout is fixed and cannot be changed.

Demos Here are some links that demonstrate the usage of the widget:

- A simple scroll container demo
- After resize the content matches the size of the scroll container.
- Content and container size can be changed. Display of scroll bars configurable.
- Content and container size can be changed. Display of scroll bars configurable.

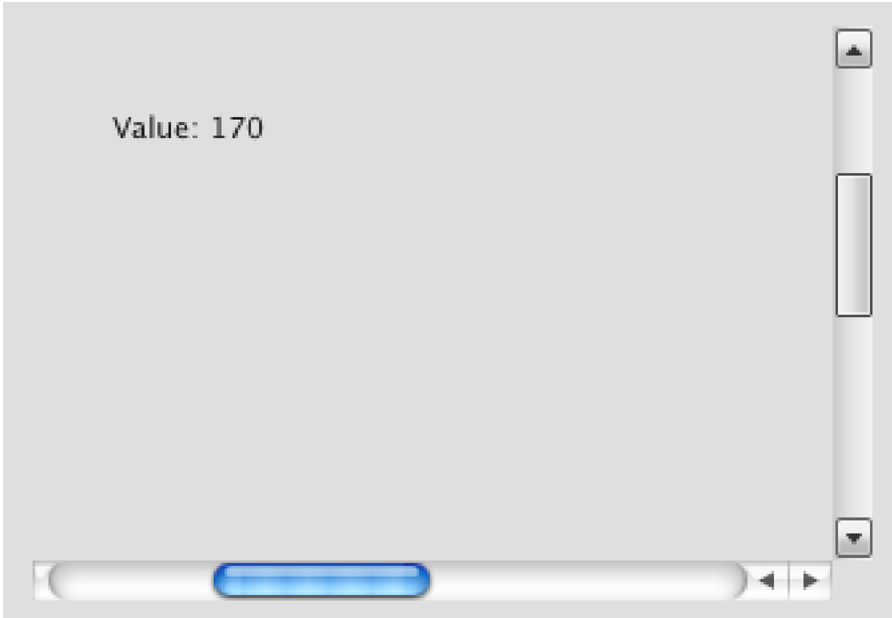
API Here is a link to the API of the Widget:
qx.ui.container.Scroll

ScrollBar The scroll bar widget exists as a custom qooxdoo scroll bar and a native browser scroll bar widget.

Which one is used as default can be controlled by the `qx.nativeScrollBars` setting.

Scroll bars are used e.g. by the *Scroll* container. Usually a scroll bar is not used directly.

Preview Image



Features

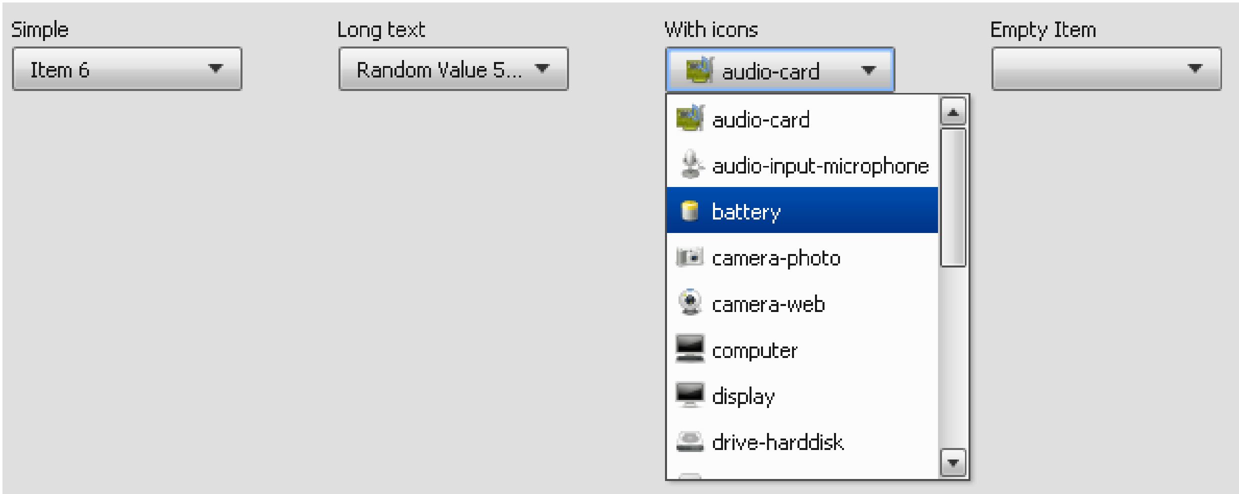
- Fully themable scroll bar (qooxdoo scroll bar)
- Size of the scroll bar knob can be adjusted

Demos Here are some links that demonstrate the usage of the widget:

- Scroll bar demo
- A simple scroll container demo

API Here is a link to the API of the Widget:
qx.ui.core.ScrollBar
qx.ui.core.NativeScrollBar

SelectBox The SelectBox has the same act like the ComboBox, but the SelectBox doesn't allow user input only selection is allowed.



Preview Image

Features

- Mouse and keyboard support.
- Items with plane text and/or icons
- Ellipsis: If the label does not fit into the widget bounds an ellipsis (”...”) is rendered at the end of the label.

Description The SelectBox has the same act like the ComboBox, but the SelectBox doesn’t allow user input only selection is allowed.

For more details about ComboBox see: *ComboBox*

Demos Here are some links that demonstrate the usage of the widget:

- SelectBox demo
- Other SelectBox demo
- Form demo

API Here is a link to the API of the Widget:
qx.ui.form.SelectBox

SlideBar The SlideBar is a container widget, which provides scrolling in one dimension (vertical or horizontal).

Preview Image

Features

- Supports `vertical` and `horizontal` orientation
- Hides the scroll buttons if the content fits into the scroll container

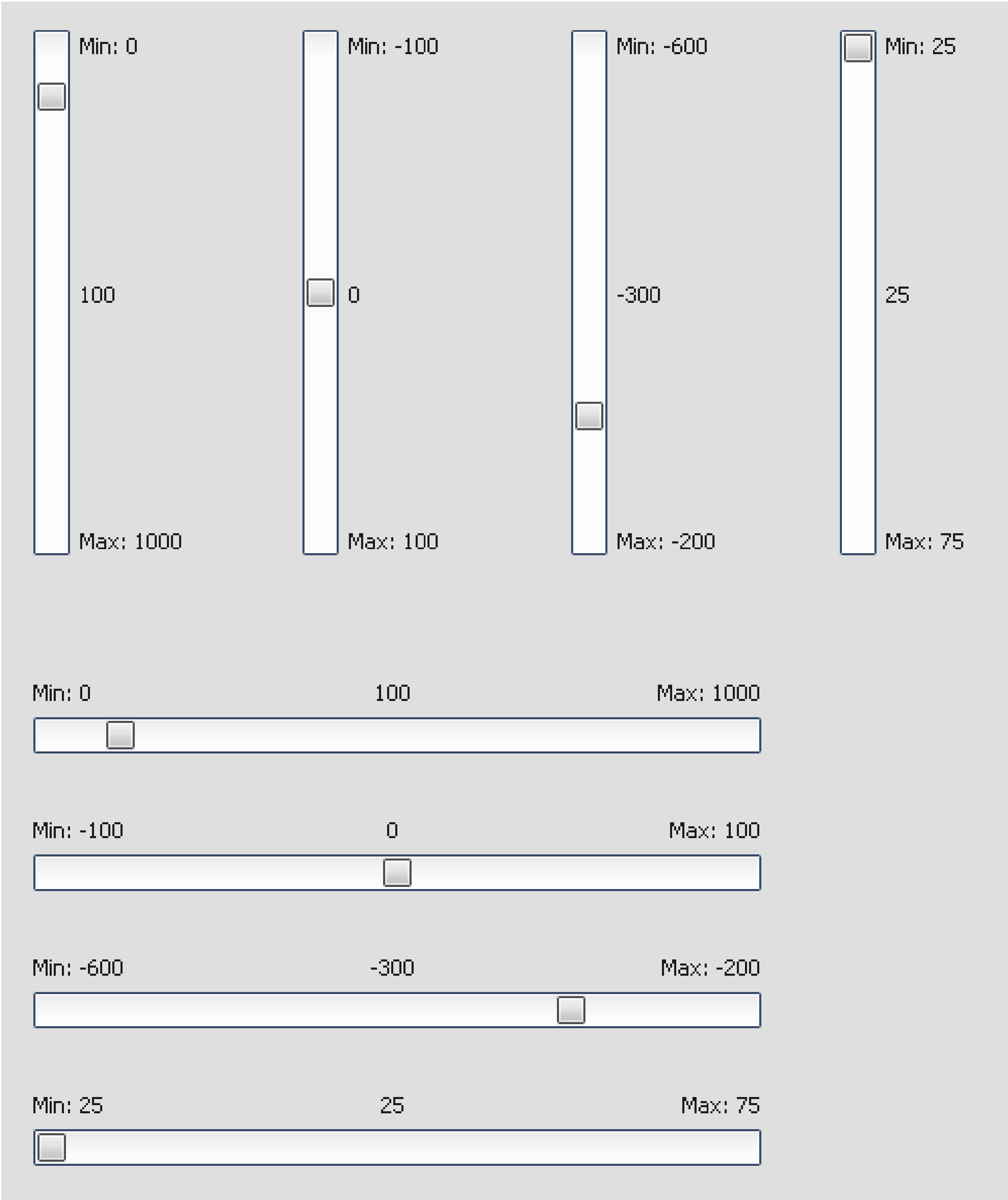
Description The SlideBar widget can be used as a replacement for a *Scroll* container if scrolling is only needed in one direction. In contrast to the Scroll container the SlideBar uses *RepeatButtons* instead of scroll bars to do the scrolling. It is used e.g. in *tab views*.

Demos Here are some links that demonstrate the usage of the widget:

- SlideBar demo

API Here is a link to the API of the Widget:
qx.ui.container.SlideBar

Slider The Slider widget is the classic widget for controlling a bounded value.



Preview Image

Features

- Mouse support.
- Horizontal and vertical orientation.

- Configurable steps.

Description The Slider widget is the classic widget for controlling a bounded value. It lets the user move a slider handle along a horizontal or vertical groove and translates the handle's position into an integer value within the defined range.

Demos Here are some links that demonstrate the usage of the widget:

- Slider demo
- Form demo

API Here is a link to the API of the Widget:
qx.ui.form.Slider

Spacer A Spacer is a "virtual" widget, which can be placed into any layout and takes the space a normal widget of the same size would take.

Features

- Spacers are invisible and very light weight because they don't require any DOM modifications

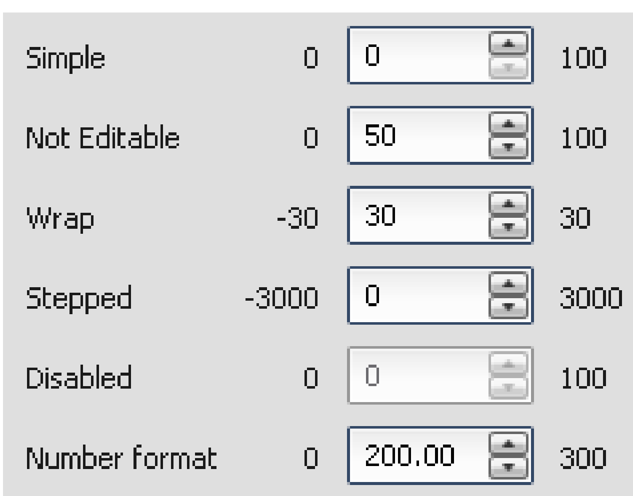
Demos Here are some links that demonstrate the usage of the widget:

- This demo shows how spacers can be used to configure vraiable spacing in a grid.
- This demo shows how spacers can be used to configure vraiable spacing in a box layout.

API Here is a link to the API of the Widget:
qx.ui.core.Spacer

Spinner A Spinner widget is a control that allows you to adjust a numerical value, typically within an allowed range e.g.: month of a year (range 1 – 12).

Preview Image



Features

- Mouse support.
- Configurable steps.
- Supports number format.

Description A spinner widget has a field to display the current value and controls such as up and down buttons to change that value. The current value can also be changed by editing the display field directly, or using mouse wheel and cursor keys.

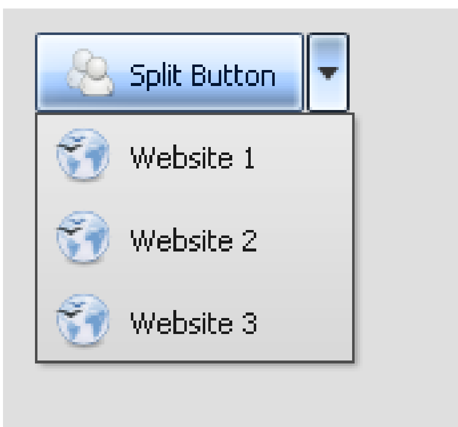
An optional NumberFormat allows you to control the format of how a value can be entered and will be displayed.

Demos Here are some links that demonstrate the usage of the widget:

- Spinner demo
- Form demo

API Here is a link to the API of the Widget:
qx.ui.form.Spinner

SplitButton The SplitButton acts like a normal button, but it opens a menu when clicking on the arrow.

Preview Image 

Features

- Contain text and/or icon.
- Mouse and keyboard support.
- Ellipsis: If the label does not fit into the widget bounds an ellipsis ("...") is rendered at the end of the label.
- Menu support.

Description The SplitButton acts like a normal button, but it opens a menu when clicking on the arrow. The menu could contain a history list or something similar.

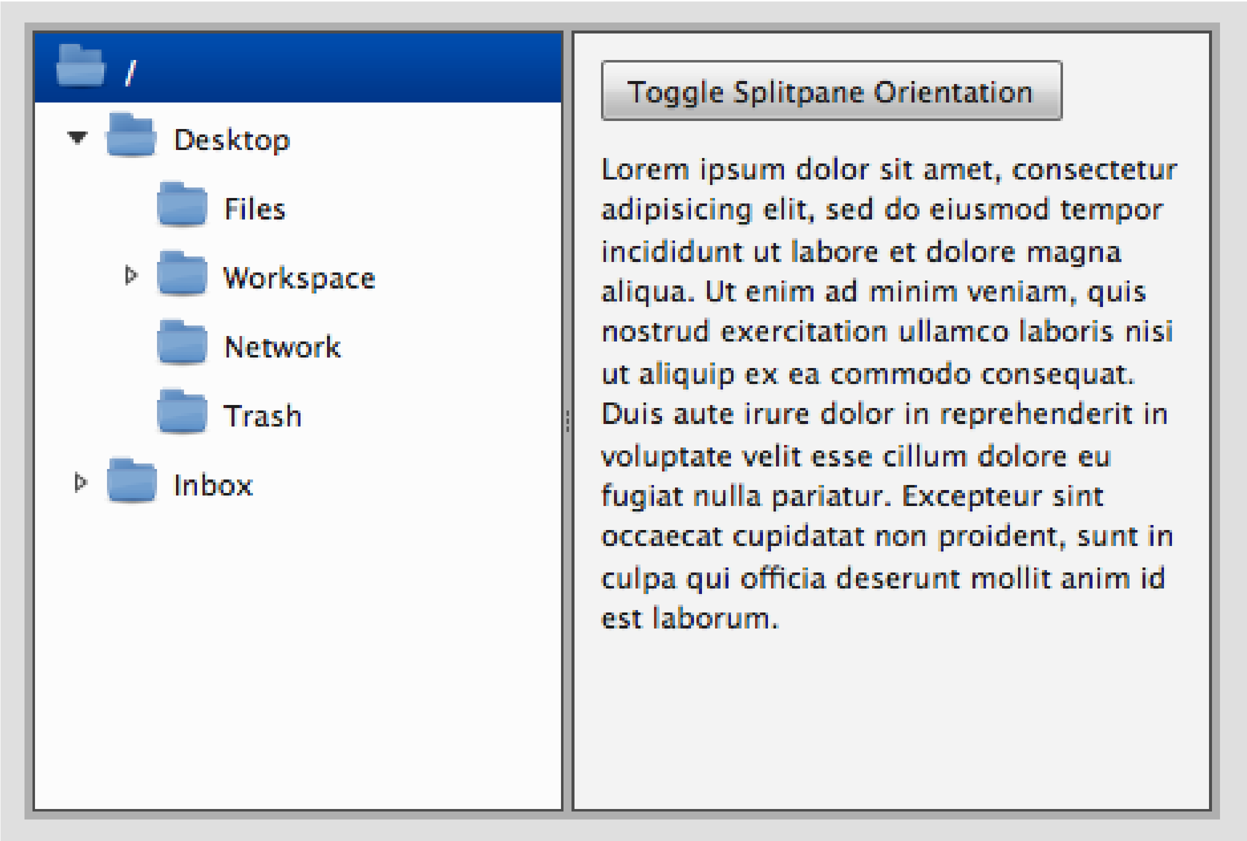
For using a menu see: *Menu*

Demos Here are some links that demonstrate the usage of the widget:

- Menu demo that contains a SplitButton
- Form demo

API Here is a link to the API of the Widget:
qx.ui.form.SplitButton

SplitPane A SplitPane is used to divide two Widgets. These widgets can be resized by clicking the splitter widget and moving the slider. The orientation property states if the widgets should be aligned horizontally or vertically.



Preview Image

Features

- Orientation
  - vertical
  - horizontal
- Autosizing with static or flex values

Description The most important class (and the class you will use mainly) inside the `qx.ui.splitpane` package is the `Pane`. One can add two widgets (of any type) to it. Besides these two widgets a `Pane` also contains a `Splitter` between them. By clicking on it (and holding down the mouse button), a `Slider` will appear and follow the mouse to indicate where the `Splitter``s will be placed when the mouse button is released.  Once the mouse button is released the available space inside the ``Pane` is redivided to both widgets according to the `Splitter`'s new position.

Demos Here are some links that demonstrate the usage of the widget:

- SplitPane that can toggle its orientation and hide/show panes

API Here is a link to the API of the Widget:
qx.ui.splitpane

Stack The stack container is a container widget, which puts its child widgets on top of each other and only the topmost widget is visible.

Features

- Two size hint modes.
  - `dynamic:true`: The stack's size is the preferred size of the visible widget
  - `dynamic:false`: The stack's size height is the to the height of the tallest widget and the stack's width is set to the width of the widest widget

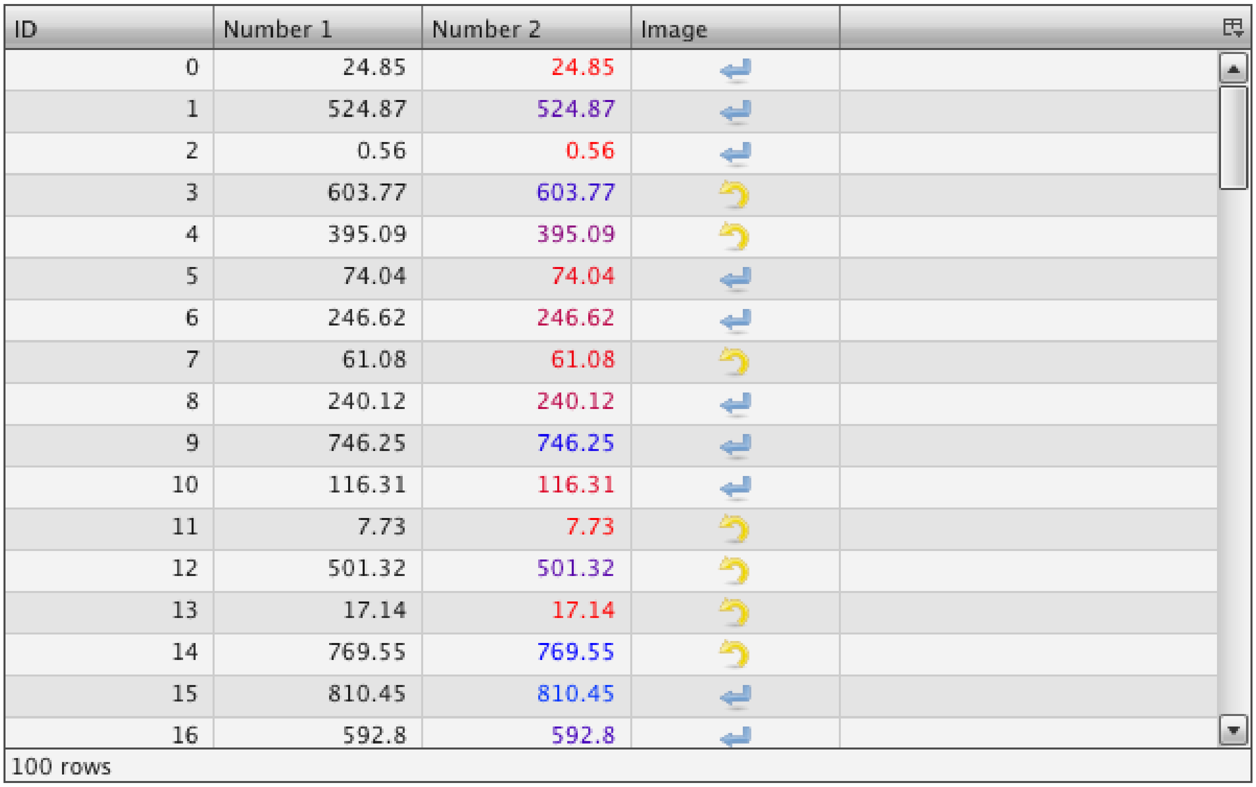
Description The stack is used if exactly one out of a collection of many widgets should be visible. This is used e.g. in the tab view widget. Which widget is visible can be controlled by using the `selected` property.

Demos Here are some links that demonstrate the usage of the widget:

- Two stack containers. The first not dynamic, the second dynamic.

API Here is a link to the API of the Widget:
qx.ui.container.Stack

Table The table package contains classes that allow you to build up virtual tables for showing data in a grid like view.



Preview Image

Description A `Table` is a widget to show a set of data in a column based view. It is based on the idea of virtual rendering. This means that only the visible rows will be rendered, which increases the performance of the widget and makes the table capable of displaying thousands of rows. But it is important to know that the table has only virtual rows, not columns. Having a huge number of columns can still decrease performance.

Features

Column Feature	Description
Display grid data	Takes an array containing an array for each row. The data in the row can be of almost any type.
Set custom header	Pre-built header renderer for icons and lables. Can be easily extended to supply a custom header cell renderer.
Column sorting	Built-in sorting accesable to the user by a click on the table header.
Reorganizing of columns	Columns can be reorganized by the user via Drag&Drop.
Change the visibility of columns	A special column visibility menu is included. It offers the user a way to show / hide single columns.
Contet menu support	The table supports content menus for each cell.
Meta Columns	You can define one or more column which have a separate scrolling if any. E.g. you could have the first column always visible, while the other columns scroll out of view.
Resizable columns	The user can resize each column individually.

Row Feature	
Render for different kinds of data types	Special renderer for boolean, dates, HTML content, numbers, passwords and strings.
Conditional rendering for individual table cells	A conditional rendere is available which can render the data in different ways dependent on the content, like applying a red text colors to negative numbers.
Row filtering	Filtering for specific data can be done with a filter method.
Virtual rendering for rows	Only the rows visible will be rendered which increases the speed of the table.
Highlight color for hovered row	The currently hovered row can be highlighted.

General Feature	
Capable of remote data gathering	A remote data model can fetch data from the server. It fetches only the current visible data which means not the whole data needs to be transfered to the client on startup.
Different selection modes	The table offers single and multi selection in different variants.
Editable cells	The cells can be set to editable. Build in editors are CheckBox, ComboBox, PasswrodField, SelectBox and TextField.
Focus indicator	The currently selected cell can have a focus indicator.
Statusbar	The table has a status bar to show the number of rows and/or custom text.

Examples

Simple The most simple table can be build in five lines of code, as you cen see in this example:

```
// table model
var tableModel = new qx.ui.table.model.Simple();
tableModel.setColumns(["ID", "A number"]);
tableModel.setData([[1, 12.23],[3, 849759438750],[2, -2]]);

// table
var table = new qx.ui.table.Table(tableModel);
this.getRoot().add(table);
```

One of the importat parts of the table is the table model. The first line creates a simple table model. In the second and third line, we configure the table model with some column names and data. With that model, we can create a table and add it to our application, as the example shows in the last two lines.

Editable Column Making for example the second column of our simple example editable can be done in one line:

```
// make second column editable
tableModel.setColumnEditable(1, true);
```

The first parameter here is the column (column numbering starts with 0), and the second one is to change the editable state.

Sorting Also adding a default sort order for the table is easy in one single line:

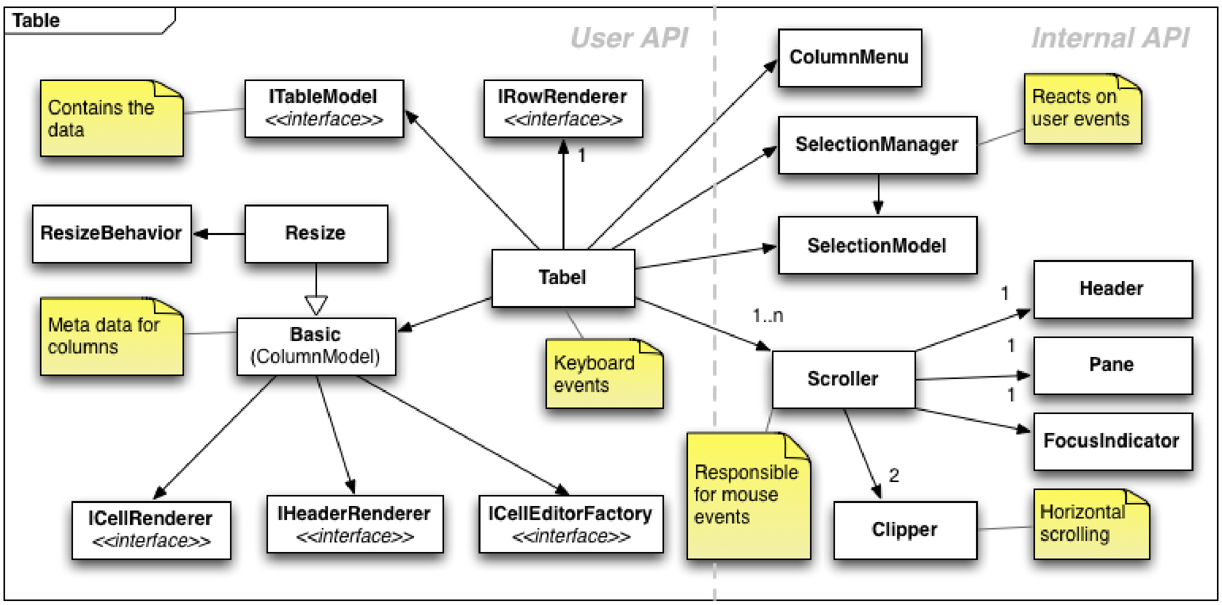
```
// sort the table on startup
tableModel.sortByColumn(0, true);
```

Again, the first argument is the column. The second argument is responsible for the sort order, `true` for ascending.

Conditional Cell Rendering As a last addition to our example we build something more complex. We want to render all negative numbers in red and all positive numbers in green:

```
// conditional rendering
var newRenderer = new qx.ui.table.cellrenderer.Conditional();
newRenderer.addNumericCondition(">", 0, null, "green");
newRenderer.addNumericCondition("<", 0, null, "red");
table.getTableColumnModel().setDataCellRenderer(1, newRenderer);
```

For that purpose, qooxdoo has a built-in conditional renderer. In the first line, we create such a renderer. The second and third line set up our conditional rules. The last line tells the table column model to use that renderere for the column with the index 1.



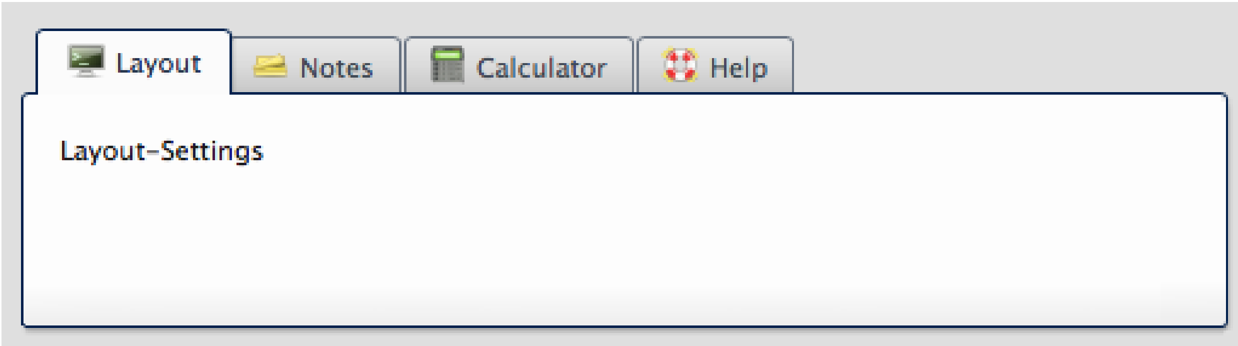
UML Diagram This diagram shows how the table uses the different kinds of classes you can find in the table namespace. The diagram is divided in two sides. The left side is interesting for the user if he wants to extend the table or wants to use its custom cell renderer for example. The right side is usually a set of internal classes the tables uses to get its general tasks done.

Further resources

- Table demos in the online Demobrowser.
- API documentation for qx.ui.table in the online APIViewer.

TabView The tab view stacks several pages above each other and allows to switch between them by using a list of buttons.

The buttons are positioned on one of the tab view's edges.



Preview Image

Features

- Tab positions: * top * bottom * left * right
- Overflow handling for tabs

Description A TabView widget consists of two parts:

- a `qx.ui.container.SlideBar` which contains a tab for every Page and can be positioned on every side of the TabView.
- a `qx.ui.container.Stack` which contains the Pages which can be added and removed at runtime.

A Page contains widgets to be shown in a TabView and usually has a label and icon to identify it.

Demos Here are some links that demonstrate the usage of the widget:

- Horizontal and vertical TabViews with a different amount of pages

API Here is a link to the API of the Widget:
qx.ui.tabview

Widget name **TODO:** *write some general stuff about the widget*

Preview Image **TODO:** *one manually taken preview image of the widget*

Features

- **TODO:** *List of features (buzzwords)*

Description **TODO:** *detailed information about the widget*

Diagram **TODO:** (if necessary) *a UML class diagram of the widget and its used classes using graphviz*

Demos Here are some links that demonstrate the usage of the widget:

- Meaningful name of the demo **TODO:** set the link
- **TODO:** *More than one demo possible ...*

API Here is a link to the API of the Widget: complete package and classname **TODO:** set the link

TextArea A TextArea some long text. The TextArea is classic GUI element.

Preview Image

Features

- Mouse and keyboard control.
- Configurable fonts and text alignment.
- Read only support.
- Automatic wrap around.

Description The TextArea is like a TextField, but for longer text input. So the TextArea supports a automatic wrap around which can be deactivated, when it is undesired.

Demos Here are some links that demonstrate the usage of the widget:

- Shows different TextArea demos

- Shows a dialog demo with an TextArea
- Show a form demo

API Here is a link to the API of the Widget:
qx.ui.form.TextArea

TextField The TextField widget is a classic GUI widget to edit text in a TextField.

Preview Image

Features

- Mouse and keyboard control.
- Configurable fonts and text alignment.
- Set maximum input length.
- Read only support.

Description The TextField widget has properties to set an alignment for the orientation and a Font for styling. Also is it possible to set the TextField read only and the maximum input length could be set.

Demos Here are some links that demonstrate the usage of the widget:

- Shows different TextField demos
- Shows a dialog demo with some TextFields
- Show a form demo
- Shows a browser demo

API Here is a link to the API of the Widget:
qx.ui.form.TextField

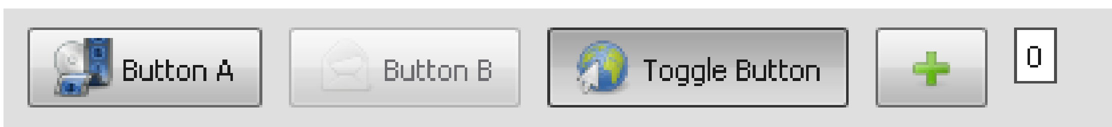
ThemedIframe **Note:** This widget is available since qooxdoo 0.8.3

Container widget for internal frames (iframes). An iframe can display any HTML page inside the widget.

Unlike `qx.ui.embed.Iframe`, which uses the browser's native iframe, ThemedIframe (particularly its scrollbars) can be visually modified according to the regular qooxdoo theming.

ToggleButton The ToggleButton widget is a classic GUI ToggleButton with to states: pressed or not pressed.

Preview Image



Features

- Contain text and/or icon.
- Mouse and keyboard support.
- Ellipsis: If the label does not fit into the widget bounds an ellipsis (”...”) is rendered at the end of the label.

Description The button is a classic GUI element, that supports two states: pressed and not pressed. The state is changed by a mouse (click) or keyboard (enter or space) event. There is an additional third state when the tri-state mode is enabled. The third state means that the widget was neither pressed nor unpressed, i.e. the state of the button is undetermined.

Demos Here are some links that demonstrate the usage of the widget:

- Button demo with all supported buttons
- Form showcase demo

API Here is a link to the API of the Widget:
qx.ui.form.ToggleButton

Toolbar The ToolBar widget is responsible for displaying a toolbar in the application. Therefore it is a container for Buttons, RadioButtons, CheckBoxes and Separators.

Preview Image

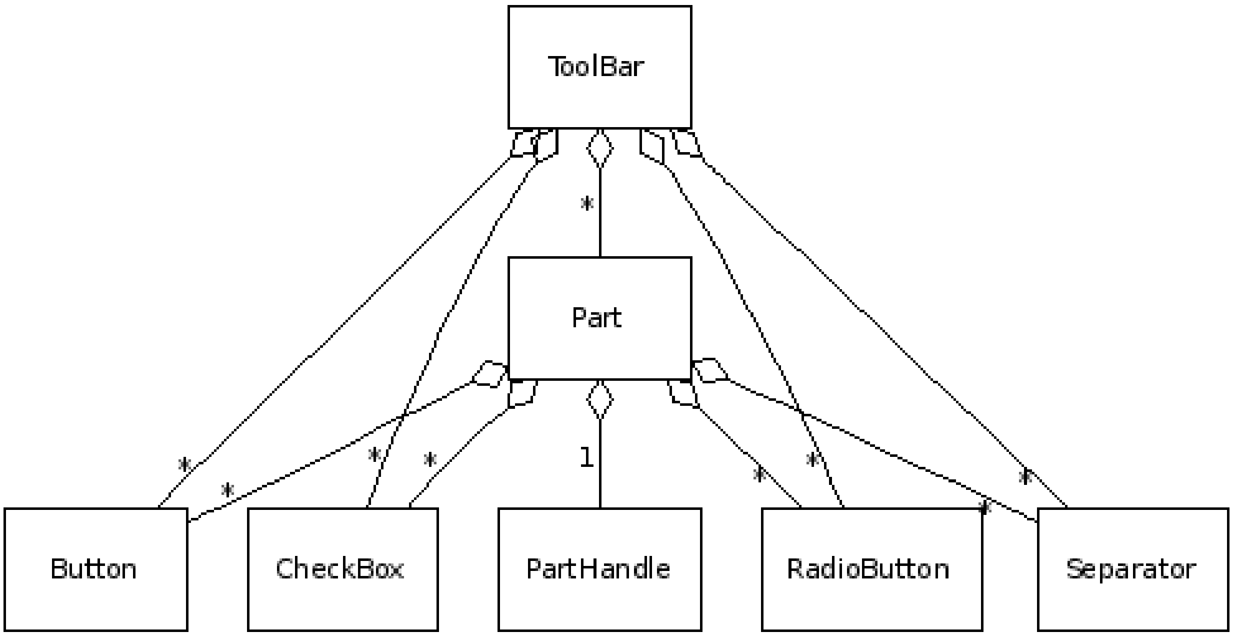
Features

- Buttons
  - Regular
  - Radio
  - Toggle
  - Menu
- Icons and / or labels for all buttons
- Separation into parts
- Separator handles

Description The qx.ui.toolbar package, which contains all stuff needed for the toolbar widget, has the main class called ToolBar. The ToolBar class is the main container for the rest of the classes. If you want to group your buttons in the toolbar, you can do this with parts. The parts class acts as a subelement of the toolbar with almost the same functionality. To a part you can add buttons. There are some kinds of buttons in the toolbar package:

- Buttons
- Radio buttons
- CheckBox buttons
- MenuButtons
- SplitButtons

These buttons can also be added directly to the toolbar if no parts are needed. For further structuring in the toolbar, a Separator is available in the package which can be added.



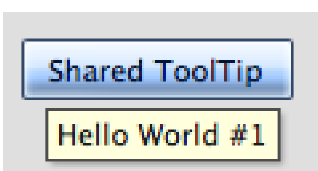
Diagram

Demos Here are some links that demonstrate the usage of the widget:

- Toolbar with all features
- Toolbar in a browser demo
- Toolbar with other menus

API Here is a link to the API of the Widget:
qx.ui.toolbar

ToolTip A Tooltip provides additional information for widgets when the user hovers over a widget. This information can consist in plain text, but also include an icon and complex HTML code.

Preview Image 

Features

- ToolTip can contain an icon
- ToolTip's label can contain HTML
- Show/hide delay

Description A ToolTip can be attached to one ore more widgets be creating a ToolTip and calling the `setToolTip()` method with the ToolTip as argument on the widget. The ToolTip will be shown as soon as the mouse overs the widget. A ToolTip can be configured to contain an icon and label and to be shown/hidden after a certain amount of time.

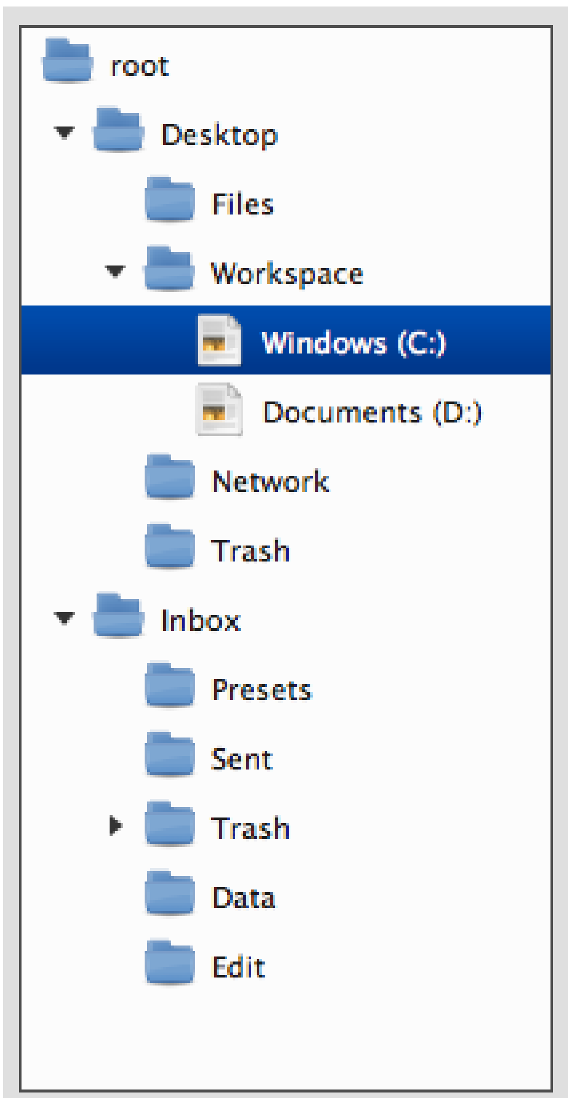
Demos Here are some links that demonstrate the usage of the widget:

- Demonstrates regular and shared ToolTips

API Here is a link to the API of the Widget:
complete package and classname

Tree The tree package contains classes that allow you to build up visual trees, like the ones you are familiar with e.g. for browsing your file system. Expanding and collapsing tree nodes is handled automatically by showing or hiding the contained subtree structure.

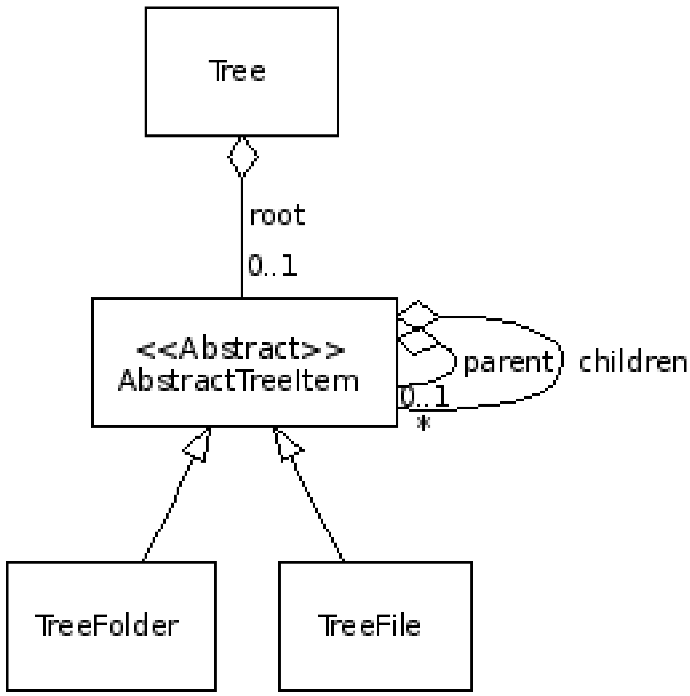
Preview Image



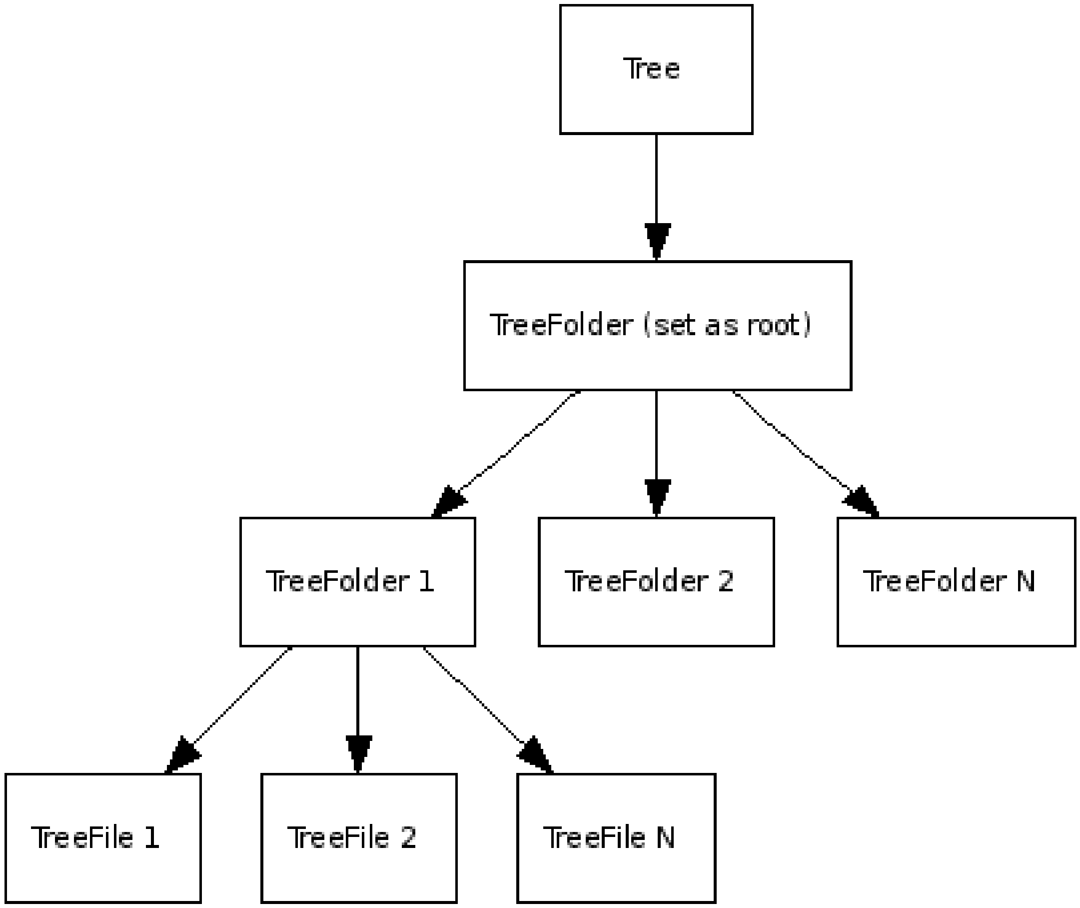
Features

- Different open and selection modes
- Toggle-able tree root

Description A `Tree` contains items in an hierarchically structure. The first item inside a `Tree` is called the root. A tree always contains one single `TreeFolder` as the root widget which itself can contain several other items. A `TreeFolder` (which is also called *node*) can contain `TreeFolder` widgets or `TreeFile` widgets. The `TreeFile` widget (also called *leaf*) consists of an icon and a label.



UML Diagram



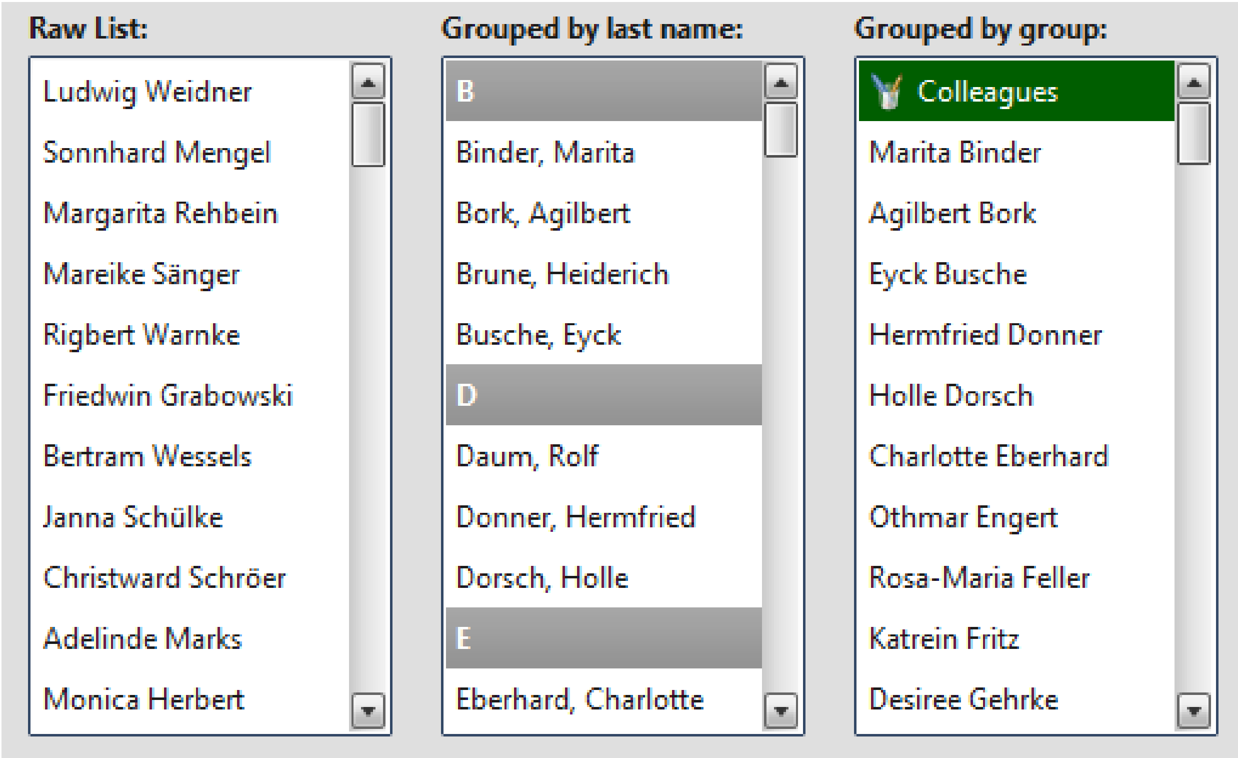
Dependencies

Demos Here are some links that demonstrate the usage of the widget:

- Complex demo which shows many features of the tree
- A multi column tree

API Here is a link to the API of the Widget:
qx.ui.tree

Virtual List The virtual List is a widget which based on the virtual infrastructure from the framework.



Preview Image

Description The `qx.ui.list.List` is based on the virtual infrastructure and supports filtering, sorting, grouping, single selection, multi selection, data binding and custom rendering.

Using the virtual infrastructure has considerable advantages when there is a huge amount of model items to render because the virtual infrastructure only creates widgets for visible items and reuses them. This saves both creation time and memory.

With the qx.ui.list.core.IListDelegate interface it is possible to configure the list's behavior (item and group renderer configuration, filtering, sorting, grouping, etc.).

Note: At the moment we only support widget based rendering for list and group items, but we are planing also to support HTML based rendering in a future release.

Code Example Here's an example. We create a simple list example with 2500 items, sorting the items ascending, selecting the 20th item and we log each selection change.

```
//create the model data
var rawData = [];
for (var i = 0; i < 2500; i++) {
  rawData[i] = "Item No " + i;
}
var model = qx.data.marshal.Json.createModel(rawData);

//create the list
var list = new qx.ui.list.List(model);

//configure the lists's behavior
var delegate = {
  sorter : function(a, b) {
```

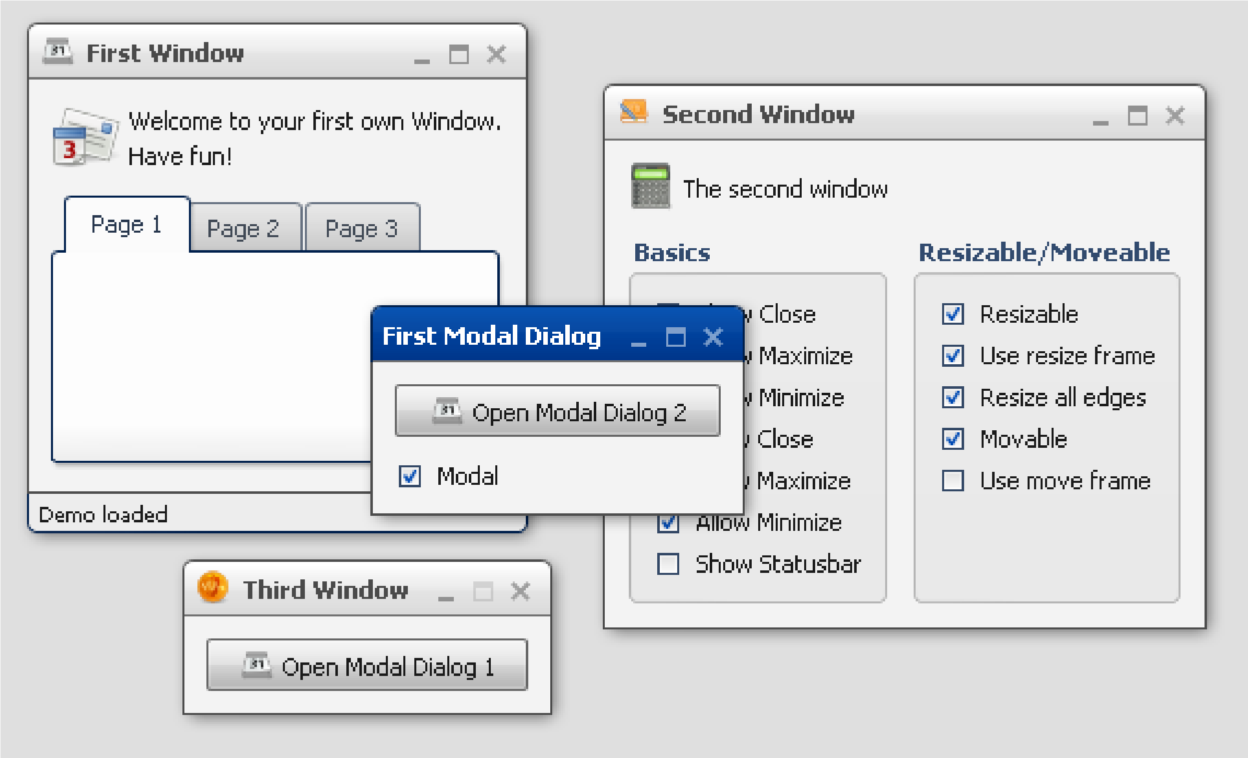
```
    return a > b ? 1 : a < b ? -1 : 0;
  }
};
list.setDelegate(delegate);

//Pre-Select "Item No 20"
list.getSelection().push(model.getItem(20));

//log selection changes
list.getSelection().addListener("change", function(e) {
  this.debug("Selection: " + list.getSelection().getItem(0));
}, this);
```

Demos Here are some links that demonstrate the usage of the widget:

- Example for the virtual List widget
- Example shows the filtering feature
- Example shows the custom rendering
- Example shows the grouping feature

API Here is a link to the API of the widget:
qx.ui.list.List

Window The window widget is similar to Windows' MDI child windows.



Preview Image

Features

- Title support text and/or icon
- Support modal window
- Status bar support
- Minimize and maximize a window
- Open and close a window
- Resize a window

Description The window widget can be used to show dialogs or to realize a MDI (Multiple Document Interface) Application.

The widgets implements all known metaphors from a window:

- minimize
- maximize
- open
- close
- and so on

The package `qx.ui.window` contains two other classes that can be used to create a MDI Application:

- The Desktop can act as container for windows. It can be used to define a clipping region for internal windows.
- The Manager handle the z-order and modality blocking of windows managed the connected desktop.

Demos Here are some links that demonstrate the usage of the widget:

- Demonstrate different window types
- Windows with using a Desktop
- A window containing a table demo
- A calculator demo

API Here is a link to the API of the Widget:
qx.ui.window.Window

11.2.2 Layout Reference

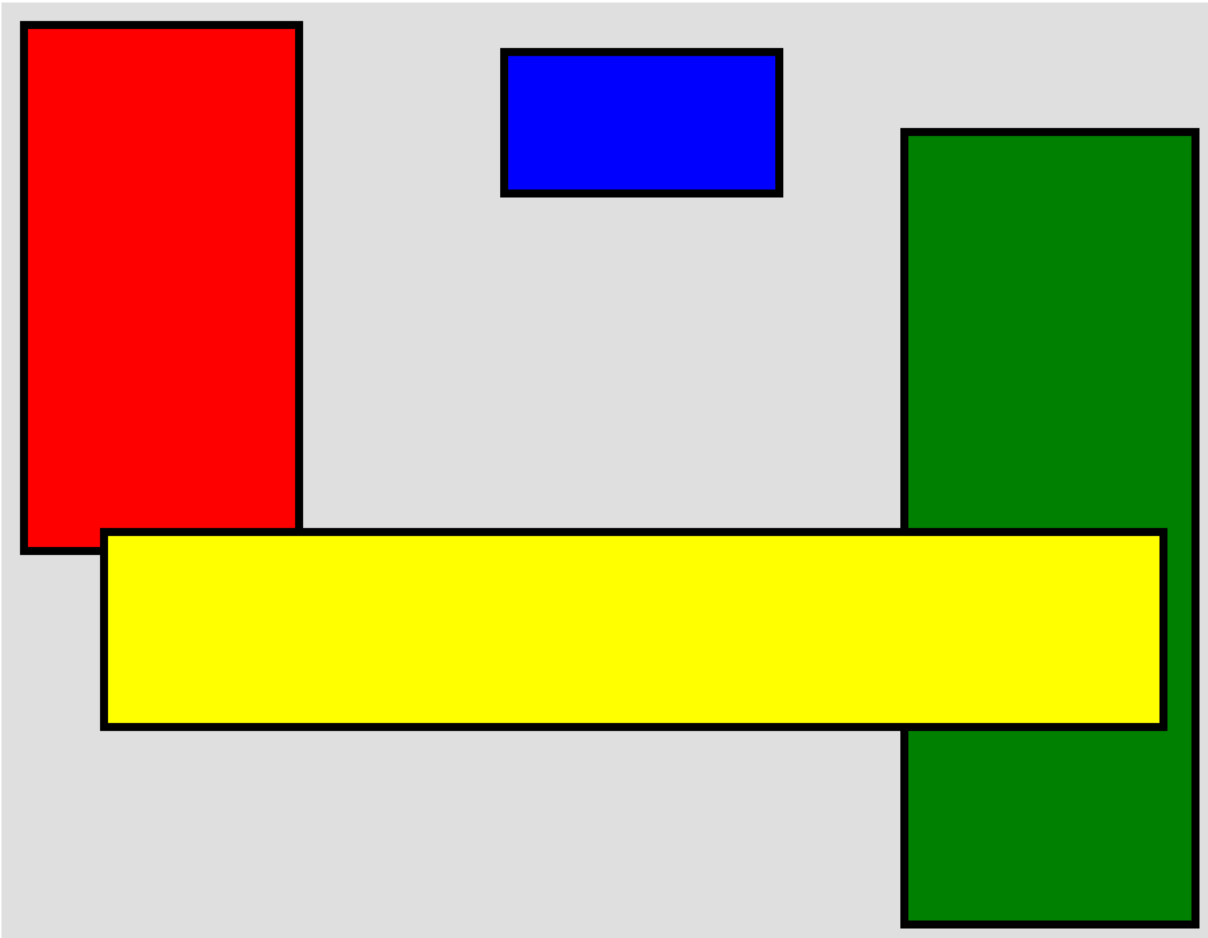
qooxdoo comes with some of the most common layout managers. The following layout managers are supported by qooxdoo:

The Basic layout is used to position the children at absolute top/left coordinates.

Basic

The Basic is used to position the children at absolute top/left coordinates.

Preview Image



Features

- Basic positioning using `left` and `top` layout properties
- Respects minimum and maximum dimensions without skrinking/growing
- Margins for top and left side (including negative margins)
- Respects right and bottom margins in the size hint
- Auto-sizing

Description

The basic layout positions each child at the coordinate given by the `left` and `top` layout properties.

The size hint of a widget configured with a Basic layout is determined such that each child can be positioned at the specified location and can have its preferred size and margin.

Margins for left and top will shift the widget position by this amount (negative values are possible). Margins for right and bottom are only respected while computing the size hint.

Layout properties

- **left**: The left coordinate in pixel (defaults to `0`)
- **top**: The top coordinate in pixel (defaults to `0`)

Alternative Names

- AbsoluteLayout (ExtJS)

Demos

Here are some links that demonstrate the usage of the layout:

- A demo of the Basic layout

API

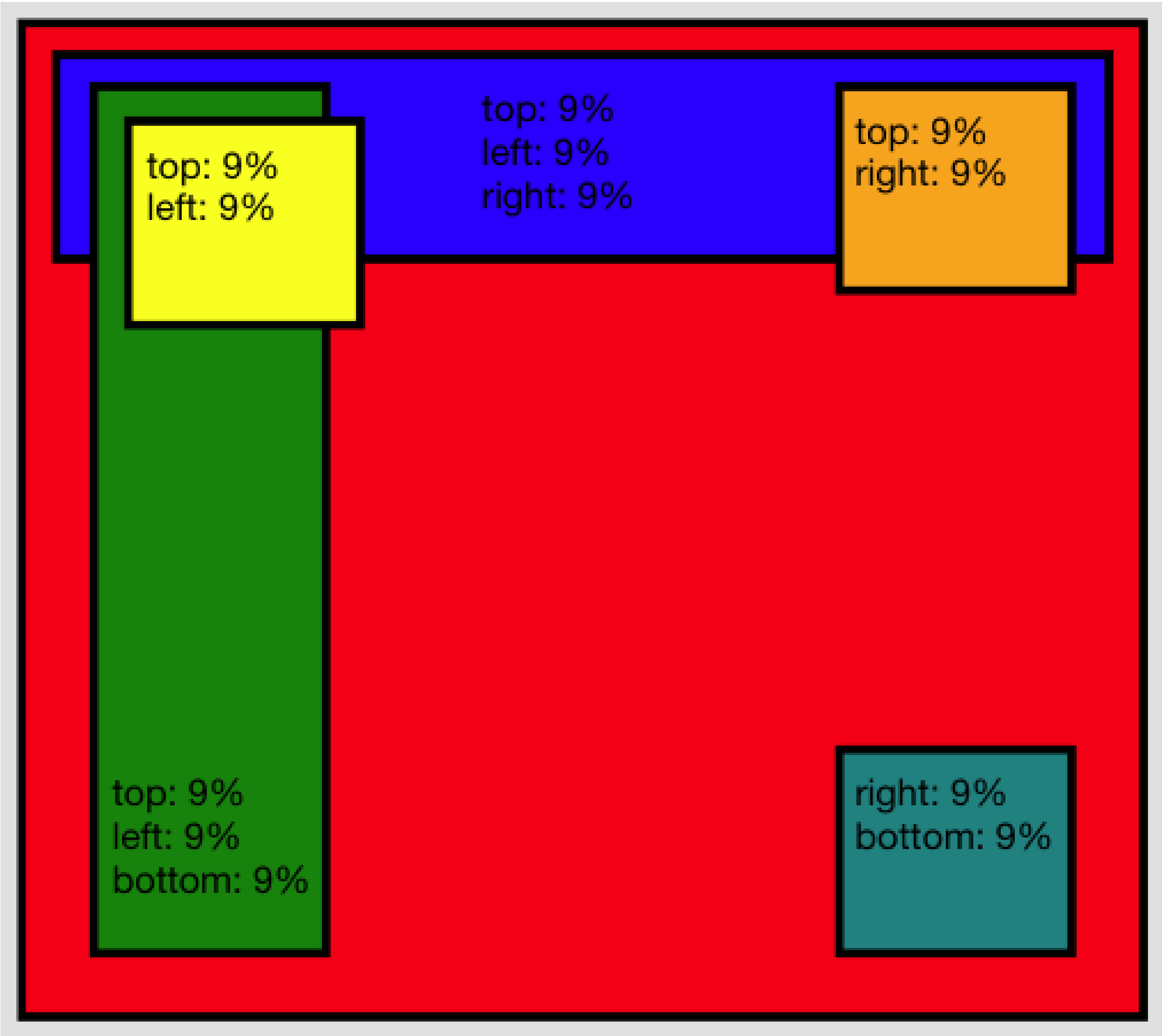
Here is a link to the API of the layout manager:
qx.ui.layout.Basic

The Canvas layout is an extended Basic layout. It is possible to position a widget relative to the right or bottom edge of the available space. The Canvas layout furthermore supports dimension and location measures in percent.

Canvas

The Canvas is an extended *Basic* layout. It is possible to position a widget relative to the right or bottom edge of the available space. The Canvas layout further has support for percent dimensions and locations.

Preview Image



Features

- Pixel dimensions and locations
- Percent dimensions and locations
- Stretching between left+right and top+bottom
- Minimum and maximum dimensions
- Children are automatically shrunk to their minimum dimensions if not enough space is available
- Auto sizing (ignoring percent values)
- Margins (also negative ones)

Description

In addition to the Basic layout the Canvas layout adds support for `right` and `bottom` layout properties. These allows to position a child in distance from the right or bottom edge of the available space. The canvas also adds support for *percent* locations and dimensions (layout properties `width` and `height`). Percents are defined as a string value (otherwise using the same layout property) with a "%" postfix.

It is possible to stretch a between the left and right edge by specifying layout properties for both `left` and `right`. The same is of cause true for `top` and `bottom`. To define a distance which is identially to each edge e.g. stretch a child to between all sides there is the `edge` property. This property accepts the same values are supported by the other location properties (including percents). Please keep in mind that often a Grow Layout might be the better choice when `edge` was planned to use in conjunction with a Canvas Layout.

The size hint of a widget configured with a Canvas layout is determined such that each child can be positioned at the specified location and can have its preferred size and margin. For this computation the layout ignores all widgets, which have a percent size or position, because These sizes depend on the actual rendered size and are not known upfront.

Layout properties

- **left** *(Integer|String)*: The left coordinate in pixel or as a percent string e.g. `20` or `30%`.
- **top** *(Integer|String)*: The top coordinate in pixel or as a percent string e.g. `20` or `30%`.
- **right** *(Integer|String)*: The right coordinate in pixel or as a percent string e.g. `20` or `30%`.
- **bottom** *(Integer|String)*: The bottom coordinate in pixel or as a percent string e.g. `20` or `30%`.
- **width** *(String)*: A percent width e.g. `40%`.
- **height** *(String)*: A percent height e.g. `60%`.

Demos

Here are some links that demonstrate the usage of the layout:

- Canvas with pixel positions
- Canvas with percent positions and dimensions
- Canvas showing left and right attachment of children
- Canvas with children having minimum and maximum dimensions

API

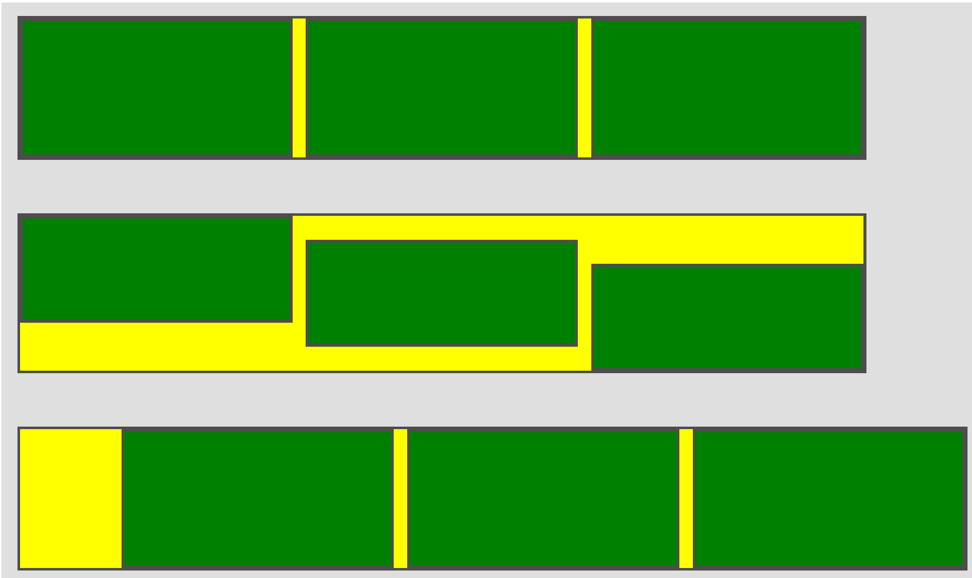
Here is a link to the API of the layout manager:
qx.ui.layout.Canvas

The Box layouts arranges their children back-to-back. The horizontal box layout arranges widgets in a horizontal row, from left to right, while the vertical box layout arranges widgets in a vertical column, from top to bottom.

HBox/VBox

The box layouts lay out their children one after the other. The horizontal box layout lays out widgets in a horizontal row, from left to right, while the vertical box layout lays out widgets in a vertical column, from top to bottom.

Preview Image



Features

- Respects Minimum and maximum dimensions
- Priorized growing/shrinking (flex)
- Margins with horizontal (HBox) resp. vertical (VBox) collapsing)
- Auto sizing (ignoring percent values)
- Percent widths (not size hint relevant)
- Alignment (Children property {@link qx.ui.core.LayoutItem#alignX} is ignored)
- Horizontal (HBox) resp. vertical (VBox) spacing (collapsed with margins)
- Property to reverse children ordering (starting from last to first)
- Vertical (HBox) resp. horizontal (VBox) children stretching (respecting size hints)

Description

Both box layouts lay out their children one after the other. This description will discuss the horizontal box layout. Everything said about the horizontal box layout applies equally to the vertical box layout just with a vertical orientation.

In addition to the child widget's own preferred width the width of a child can also be defined as *percent* values. The percent value is relative to the inner width of the parent widget without any spacings. This means a horizontal box layout with two children of width `50%` and with a spacing will fit exactly in the parent.

The horizontal box layout tries to stretch all children vertically to the height of the box layout. This can be supressed by setting the child property allowGrowY to false. If a child is smaller than the layout and cannot be stretched it will be aligned according to its alignY value. The alignX property of the layout itself defines the horizontal alignment of all the children as a whole.

The horizontal spacing can be defined using the property spacing. In addition to the spacing property each widget can define left and a right `margin`. Margins and the spacing are always collapsed to the largest single value. It for example the layout has a spacing of `10` pixel and two consecutive child widgets A and B - A with a right margin of `15` and B with a left margin of `5` - than the spacing between these widgets would be `15`, the maximum of these values.

The preferred height of an horizontal box layout is determined by the highest child widged. The preferred with is the sum of the widths of each child plus the spacing resulting from margins and the `spacing` property.

Layout properties

- **flex** *(Integer)*: Defines the flexibility (stretching factor) of the child (defaults to `0`)
- **width** *(String)*: Defines a percent width for the item. The percent width, when specified, is used instead of the width defined by the size hint. The minimum and maximum width still takes care of the elements limitations. It has no influence on the layout's size hint. Percents are mainly useful for widgets which are sized by the outer hierarchy.

Alternative Names

- QVBoxLayout (Qt)
- StackPanel (XAML)
- RowLayout (SWT)

Demos

Here are some links that demonstrate the usage of the layout:

- Simple HBox usage
- HBox with flex widths
- HBox with child margins
- HBox with percent widths
- HBox with switchable ''reversed'' property
- HBox with separators
- HBox with vertical shrinking
- Simple VBox usage
- VBox with flex heights
- VBox with child margins
- VBox with percent heights
- VBox with switchable ''reversed'' property
- VBox with separators
- VBox with horizontal shrinking

API

Here is a link to the API of the layout manager: qx.ui.layout.HBox
qx.ui.layout.VBox

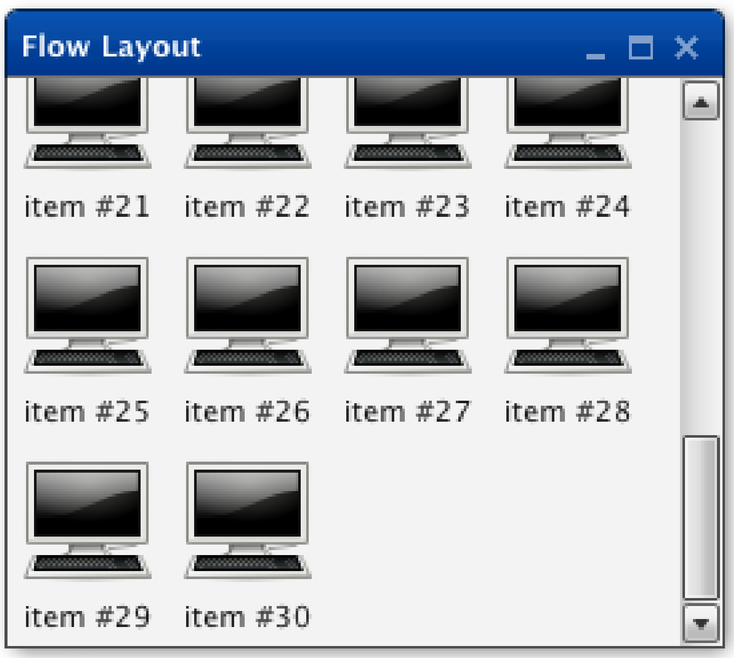
The Flow layout places widget next to each other from left to right. If the available width is not sufficient an automatic line break is inserted.

Flow

Note: This layout manager is available since qooxdoo 0.8.3.

A basic layout, which supports positioning of child widgets in a 'flowing' manner, starting at the container's top/left position, placing children left to right (like a HBox) until the there's no remaining room for the next child. When out of room on the current line of elements, a new line is started, cleared below the tallest child of the preceeding line – a bit like using 'float' in CSS, except that a new line wraps all the way back to the left.

Preview Image



This image shows a gallery implemented using a Flow layout.

Features

- Reversing children order
- Manual line breaks
- Horizontal alignment of lines
- Vertical alignment of individual widgets within a line

- Margins with horizontal margin collapsing
- Horizontal and vertical spacing
- Height for width calculations
- Auto-sizing

Description

The Flow layout imitates the way text is rendered. Each child is placed horizontally next to each other. If the remaining space is too small a new line is created and the child is placed at the start of the new line.

It is possible to specify a horizontal alignment for all children. This is equivalent to `center`, `left` or `right` alignment of text blocks. Further it is possible to specify the vertical alignment of each child in a line.

This layout supports `height for width`, which means that given a fixed width it can calculate the required height.

Layout properties

- **lineBreak** *(Boolean)*: If set to `true` a forced line break will happen after this child widget.

Demos

Here are some links that demonstrate the usage of the layout:

- Flow layout demo

API

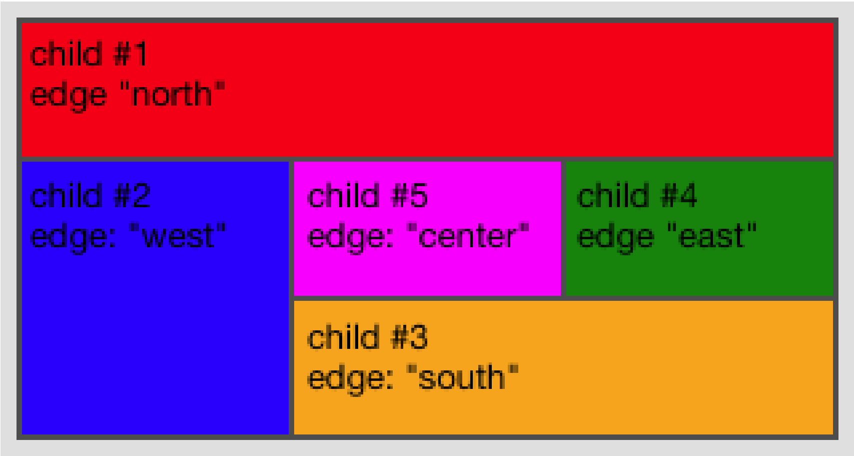
Here is a link to the API of the layout manager:
qx.ui.layout.Flow

A Dock layout attaches the children to the edges of the available space.

Dock

Docks children to one of the edges.

Preview Image



Features

- Percent width for left/right/center attached children
- Percent height for top/bottom/center attached children
- Minimum and maximum dimensions
- Prioritized growing/shrinking (flex)
- Auto sizing
- Margins and Spacings
- Alignment in orthogonal axis (e.g. alignX of north attached)
- Different sort options for children

Description

The `Dock` layout attaches the children to the edges of the availble space. The space distribution respects the child order and starts with the first child. Every added child reduces the available space of the other ones. This is important because for example a left attached child reduces the available width for top attached children and vice-versa. This layout is mainly used for the basic application layout structure.

Layout properties

- **edge** *(String)*: The edge where the layout item should be docked. This may be one of `north`, `east`, `south`, `west` or `center`. (Required)
- **width** *(String)*: Defines a percent width for the item. The percent width, when specified, is used instead of the width defined by the size hint. This is only supported for children added to the north or south edge or are centered in the middle of the layout. The minimum and maximum width still takes care of the elements limitations. It has no influence on the layout's size hint. Percents are mainly useful for widgets which are sized by the outer hierarchy.

- **height** *(String)*: Defines a percent height for the item. The percent height, when specified, is used instead of the height defined by the size hint. This is only supported for children added to the west or east edge or are centered in the middle of the layout. The minimum and maximum height still takes care of the elements limitations. It has no influence on the layout's size hint. Percents are mainly useful for widgets which are sized by the outer hierarchy.

Alternative Names

- BorderLayout (Qt)
- DockPanel (XAML)
- BorderLayout (Java)
- BorderLayout (ExtJS)

Demos

Here are some links that demonstrate the usage of the layout:

- Simple docks
- Docks with auto sizing and spacings
- Docks with flex sizes (growing)
- Docks with flex sizes (shrinking)
- Docks with child margins
- Docks with percent sizes
- Docks with separators

API

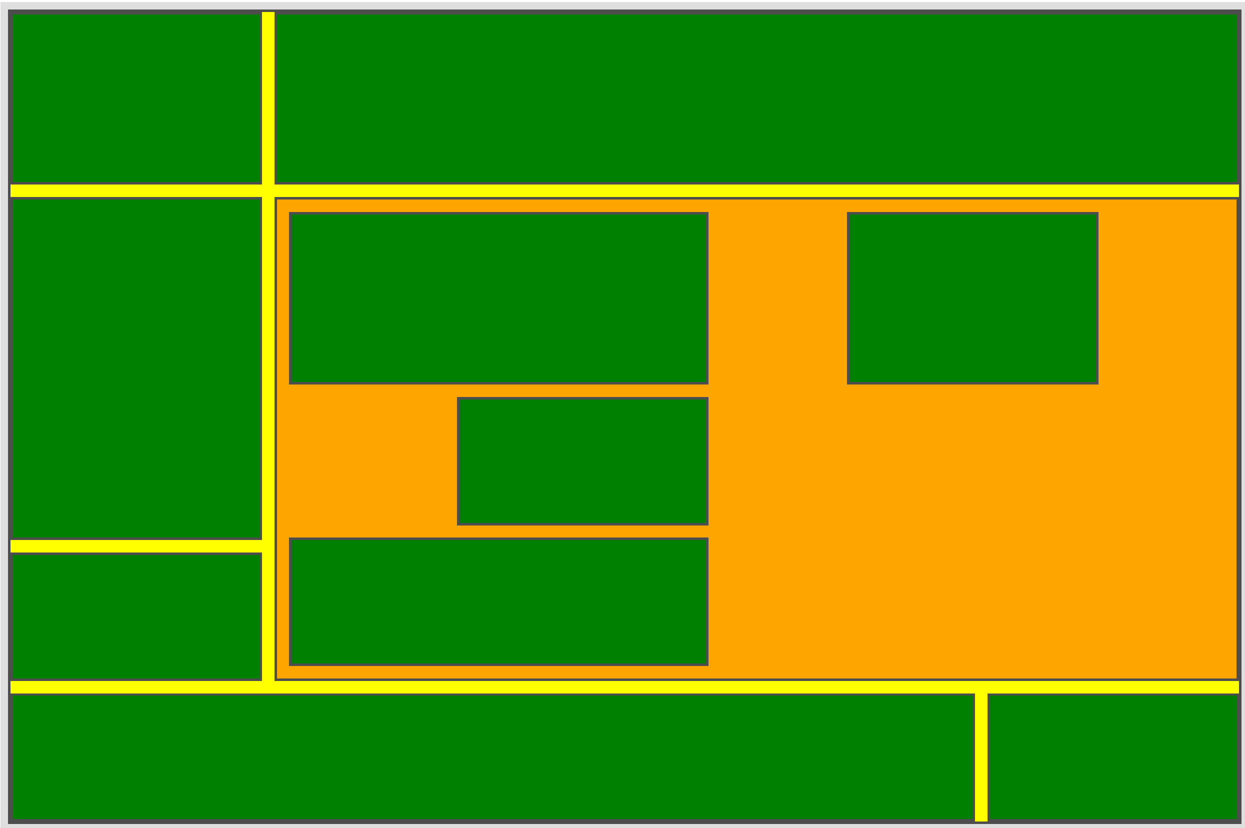
Here is a link to the API of the layout manager:
qx.ui.layout.Dock

The Grid layout arranges items in a two dimensional grid. Widgets can be placed into the grid's cells and may span multiple rows and columns.

Grid

The grid layout manager arranges the items in a two dimensional grid. Widgets can be placed into the grid's cells and may span multiple rows and columns.

Preview Image



This image show two nested grids with column and row spans.

Features

- Flex values for rows and columns
- Minimal and maximal column and row sizes
- Manually setting of column and row sizes
- Horizontal and vertical alignment
- Horizontal and vertical spacing
- Column and row spans
- Auto-sizing

Description

The grid arranges the child widgets in a two dimensional grid. Each child is associated with a grid `column` and `row`. Widgets can span multiple cells by setting the `colSpan` and `rowSpan` layout properties. However each grid cell can only contain one widget. Thus child widgets can never overlap.

The grid computes the preferred with/height of each column/row based on the preferred size of the child widgets. The computed column widths and row heights can be overridden by explicitly setting them using setColumnWidth and setRowHeight. Minimum and maximum sizes for columns/rows can be set as well.

By default no column or row is stretched if the available space is larger/smaller than the needed space. To allow certain rows/columns to be stretched each row/column can have a *flex* value.

Layout properties

- **row** *(Integer)*: The row of the cell the widget should occupy. Each cell can only contain one widget. This layout property is mandatory.
- **column** *(Integer)*: The column of the cell the widget should occupy. Each cell can only contain one widget. This layout property is mandatory.
- **rowSpan** *(Integer)*: The number of rows, the widget should span, starting from the row specified in the `row` property. The cells in the spanned rows must be empty as well. (defaults to `1`)
- **colSpan** *(Integer)*: The number of columns, the widget should span, starting from the column specified in the `column` property. The cells in the spanned columns must be empty as well. (defaults to `1`)

Alternative Names

- QGridLayout (Qt)
- Grid (XAML)
- TableLayout (ExtJS)

Demos

Here are some links that demonstrate the usage of the layout:

- Simple grids
- Complex grids
- A grid with different cell alignments
- An animated grid

API

Here is a link to the API of the layout manager:
qx.ui.layout.Grid

The Grow layout stretches all children to the full available size but still respects limits configured by min/max values.

Grow

The grow layout stretches all children to the full available size but still respects limits configured by min/max values.

Features

- Auto-sizing
- Respects minimum and maximum child dimensions

Description

The Grow layout is the simplest layout in qooxdoo. It scales every child to the full available width and height (still respecting limitations of each child). It will place all children over each other with the top and left coordinates set to `0`. This layout is usually used with only one child in scenarios where exactly one child should fill the whole content (e.g. adding a TabView to a Window). This layout performs a lot better in these cases than for example a canvas layout with `edge=0`.

Layout properties

The Grow layout does not have any layout properties.

Alternative Names

- FitLayout (ExtJS)

API

Here is a link to the API of the layout manager:
qx.ui.layout.Grow

There are a few more layouts bundled with the default qooxdoo distribution but those are mostly intended for use by a specific component. For example the *Atom* uses the Atom Layout, the *SplitPane* uses the two split layouts HLayout and VLayout.

Through the simple API it should be quite easy to write custom layouts if the included ones do not meet demands. Simply derive from the Abstract layout and start with a refined version of the method renderLayout().

11.3 Tooling

11.3.1 Default Generator Jobs

This page describes the jobs that are automatically available to all skeleton-based applications (particularly, applications with config.json files that include the framework's *application.json* config file). Mainly this is just a reference list with short descriptions of what the jobs do. But in some cases, there is comprehensive documentation about the interface of this job and how it can be parametrized (This would usually require changing your *config.json* configuration file).

Action Jobs

These jobs can be invoked with the generator, e.g. as `generate.py <jobname>`.

api

Create api doc for the current library. Use the following macros to tailor the scope of classes that are going to show up in the customized apiviewer application:

```
"API_INCLUDE" = ["<class_patt1>", "<class_patt2>", ...]
"API_EXCLUDE" = ["<class_patt1>", "<class_patt2>", ...]
```

The syntax for the class pattern is like those for the *include* config key.

api-data

Create the api data for the current library. This is included in the *api* job, but allows you to re-generate the api data *.json* files for the classes without re-generating the Apiviewer application as well. Moreover, you can supply class names as command line arguments to only re-generate the api data for those:

```
sh> generate.py api-data my.own.ClassA ...
```

Beware though that in such a case the tree information provided to the Apiviewer (i.e. what you see in the Apiviewer's tree view on the left) is also restricted to those classes (augmented by stubs for their ancestors for hierarchy resolution). But this should be fine for developing API documenation for specific classes.

build

Create build version of current application.

clean

Remove local cache and generated .js files (source/build).

distclean

Remove the cache and all generated artefacts of this library (source, build, ...).

fix

Normalize whitespace in .js files of the current library (tabs, eol, ...).

info

Running this job will print out various information about your setup on the console. Information includes your qooxdoo and Python version, whether source and/or build version of your app has been built, stats on the cache, asf.

inspector

Create an instance of the Inspector in the current application. The inspector is a debugging tool that allows you to inspect your custom application while running. You need to run the *source* job first, the run the *inspector* job. You will then be able to open the file `source/inspector.html` in your browser. The source version of your application will be loaded into the inspector automatically.

lint

Check the source code of the .js files of the current library.

migration

Migrate the .js files of the current library to the current qooxdoo version.

Running the migration job Here is a sample run of the migration job:

```
./generate.py migration

NOTE:    To apply only the necessary changes to your project, we
         need to know the qooxdoo version it currently works with.

Please enter your current qooxdoo version [1.0] :
```

Enter your qooxdoo version or just hit return if you are using the version given in square brackets.

```
MIGRATION SUMMARY:

Current qooxdoo version:   1.0
Upgrade path:              1.0.1 -> 1.1 -> 1.2

Affected Classes:
    feedreader.view.Header
    feedreader.view.Article
    feedreader.view.Tree
    feedreader.PreferenceWindow
    feedreader.view.ToolBar
    feedreader.FeedParser
    feedreader.view.Table
    feedreader.Application
    feedreader.test.DemoTest

NOTE:    It is advised to do a 'generate.py distclean' before migrating any files.
         If you choose 'yes', a subprocess will be invoked to run distclean,
         and after completion you will be prompted if you want to
         continue with the migration. If you choose 'no', the distclean
         step will be skipped (which might result in potentially unnecessary
         files being migrated).

Do you want to run 'distclean' now? [yes] :
```

Enter "yes".

```
WARNING: The migration process will update the files in place. Please make
         sure, you have a backup of your project. The complete output of the
         migration process will be logged to 'migration.log'.

Do you want to start the migration now? [no] :
```

Enter "yes".

Check `migration.log` for messages that contain *foo.js has been modified. Storing modifications ...* to verify changes to class code.

simulation-build

Creates a runner application (the *Simulator (Experimental)*) for Selenium-based GUI interaction tests of the current library.

The Simulator is configured using the "*settings*" key of this job. The following settings are supported:

- **simulator.testBrowser** (String, default: `*firefox3`)
  - A browser launcher as supported by Selenium RC (see the Selenium documentation for details).
- **simulator.autHost** (String, default: `http://localhost`)
  - Protocol and host name that Selenium should use to access the application to be tested
- **simulator.autPath** (String, default: `/<applicationName>/source/index.html`)
  - Server path of the tested application.
- **simulator.selServer** (String, default: `localhost`)
  - Host name of the machine running the Selenium RC server instance to be used for the test.
- **simulator.selPort** (Integer, default: `4444`)
  - Number of the port the Selenium RC server is listening on
- **simulator.globalErrorLogging** (Boolean, default: `false`)
  - Log uncaught exceptions in the AUT.
- **simulator.testEvents** (Boolean, default: `false`)
  - Activate AUT event testing support.
- **simulator.applicationLog** (Boolean, default: `false`)
  - Capture the AUT's log output.

simulation-run

Starts Rhino and executes a *Simulator (Experimental)* test application generated by `simulation-build`.

Configured using the "*simulate*" key.

pretty

Pretty-formatting of the source code of the current library.

source

Create source version of current application.

source-all

Create source version of current application, with all classes.

test

Create a test runner app for unit tests of the current library.

- Use the following macro to tailor the scope of classes in which unit test classes are searched for:

  ```
  "TEST_INCLUDE" = ["<class_patt1>", "<class_patt2>", ...]
  ```

  The syntax for the class pattern is like those for the *include* config key.

- The libraries from the *libraries* job will be included when building the test application (the application containing your unit tests is a separate application which is loaded into the runner application).

- If you want to break out from the reliance on the *libraries* job altogether, or have very specific settings that must be applied to the test application, you can provide a custom includer job *common-tests* which may contain a custom *library* key and other keys. But then you have to make sure it contains the Testrunner library as well.

  ```
  "common-tests" :
  {
    "extend"    : [ "libraries" ],

    "let" :      { "LOCALES" : ["de", "de_DE", "fr", "fr_FR" ] },

    "library" :
    [
      { "manifest" : "${QOOXDOO_PATH}/framework/Manifest.json" },
      { "manifest" : "${TESTRUNNER_ROOT}/Manifest.json" }
    ],

    "include" : ["testrunner.TestLoader", "${TEST_INCLUDE}", "${QXTHEME}"],

    "settings" :
    {
      "qx.theme" : "${QXTHEME}",
      "qx.globalErrorHandling" : "on"
    },

    "cache" :
    {
      "compile" : "${CACHE}"
    }
  }
  ```

  This allows you to tailor most of the parameters that influence the creation of the test application.

test-source

Create a test runner app for unit tests (source version) of the current library.

The same customization interface applies as for the default *test* job.

test-inline

Create an inline test runner app for unit tests of the current library.

The same customization interface applies as for the default *test* job.

test-native

Create a native test runner app for unit tests of the current library.

The same customization interface applies as for the default *test* job.

translation

Create .po files for current library.

Includer Jobs

These jobs don't do anything sensible on their own, so it is no use to invoke them with the generator. But they can be used in the application's `config.json`, to modify the behaviour of other jobs, as they pick up their definitions.

common

Common includer job for many default jobs, mostly used internally. You should usually not need to use it; if you do, use with care.

libraries

This job should take a single key, *library*. The *libraries* job is filled by default with your application and the qooxdoo framework library, plus any additional libraries you specify in a custom *libraries* job you added to your application's *config.json*. Here, you can add additional libraries and/or contributions you want to use in your application. See the linked reference for more information on the library key. Various other jobs will evaluate the *libraries* job (like *api*, *test*), to work on a common set of libraries.

```
"libraries" :
{
  "library" : [ { "manifest" : "some/other/lib/Manifest.json" }]
}
```

profiling

Includer job, to activate profiling.

log-parts

Includer job, to activate verbose logging of part generation; use with the -v command line switch.

log-dependencies

Includer job, to activate verbose logging of class dependencies; use with the -v command line switch.

11.3.2 Reference Listing of Config Keys

This page contains the complete list of configuration keys and their sub-structures.

Mandatory keys in a context are marked *'(required)'*, all other keys can be considered optional (most have default values). Special note boxes starting with *'peer-keys'* indicate interactions of the current key with other configuration keys that should be present in the job for the current key to function properly. E.g. the key *compile* will use the peer-key *cache* in the job definition for its workings. Again, in many cases fall-back defaults will be in place, but relying on them might lead to sub-optimal results.

add-script

Add a pre-fabricated JS file to the application. Takes a list.

```
"add-script" :
[
  {
    "uri" : "<script-uri>"
  }
]
```

Note: peer-keys: *compile*

- **uri** *(required)* : URI with which the script will be loaded, relative to the index.html.

api

Triggers the generation of a custom Apiviewer application. Takes a map.

```
"api" :
{
  "path"   : "<path>",
  "verify" : [ "links" ]
}
```

Note: peer-keys: *cache*, *include*, *library*

- **path** *(required)* : Path where the Apiviewer application is to be stored, relative to the current directory.
- **verify** : Things to check during generation of API data.
  - **links** : Check internal documentation links (@link{...}) for consistency.

asset-let

Defines macros that will be replaced in #asset hints. Takes a map.

```
"asset-let" :
{
  "<macro_name>" : [ "foo", "bar", "baz" ]
}
```

Each entry is

- <macro_name> : [<list of replacement strings>] Like with macros, references (through '${macro_name}') to these keys in #asset hints in source files will be replaced. Unlike macros, each listed value will be used, and the result is the list of all ensuing expressions, so that all resulting assets will be honored.

Special section

cache

Define the paths to cache directories, particularly to the compile cache. Takes a map.

```
"cache" :
{
  "compile"     : "<path>",
  "downloads"   : "<path>",
  "invalidate-on-tool-change" : (true|false)
}
```

Possible keys are

- **compile** : path to the "main" cache, the directory where compile results are cached, relative to the current (default: "*${CACHE}*")
- **downloads** : directory where to put downloads (e.g. `contrib://*` libraries), relative to the current (default: "*${CACHE}*/downloads")
- **invalidate-on-tool-change** : when true, the *compile* cache (but not the downloads) will be cleared whenever the tool chain is newer (relevant mainly for trunk users; default: *true*)

Special section

clean-files

Triggers clean-up of files and directories within a project and the framework, e.g. deletion of generated files, cache contents, etc. Takes a map.

```
"clean-files" :
{
  "<doc_string>" :
  [
    "<path>",
    "<path>"
  ]
}
```

Note: peer-keys: *cache*

Each key is a doc string that will be used in logging when deleting the corresponding files.

- <doc_string> : arbitrary string

- <path> : file/path to be deleted; may be relative to config file location; *file globs* allowed

collect-environment-info

Triggers the collection of information about the qooxdoo environment, and prints it to the console. Takes a map.

```
"collect-environment-info" : {}
```

Note: peer-keys: *cache*

This key currently takes no subkeys, but you still have to provide an empty map. The information collected includes the qooxdoo version, the Python version, the path to the cache, stats about the cache contents, whether the current application has been built, asf.

combine-images

Triggers the creation of combined image files that contain various other images. Takes a map. *This action key requires an external program (ImageMagic) to run successfully.*

```
"combine-images" :
{
  "images" :
  {
    "<output_image>" :
    {
      "prefix": [ "<string>", "<altstring>" ],
      "layout": ("horizontal"|"vertical"),
      "input" :
      [
        {
          "prefix" : [ "<string>", "<altstring>" ],
          "files"  : [ "<path>", "<path>" ]
        }
      ]
    }
  }
}
```

Note: peer-keys: *cache*

- **images** : map with combine entries
  - <output_image> : path of output file; may be relative to the config file location
    * **prefix** *(required)*: takes a list; the first element is a prefix of the path given in <output_image>, leading up to, but not including, the library name space of the output image; this prefix will be stripped from the ouput path, and will be replaced by an optional second element of this setting, to eventually obtain the image id of the output image;
    * **layout** : either "horizontal" or "vertical"; defines the layout of images within the combined image (default: "horizontal")
    * **input** *(required)*: list of groups of input files, each group sharing the same prefix; each group consists of:
      · **prefix** *(required)*: takes a list; analogous to the *prefix* attribute of the ouput image, the first element of the setting will be stripped from the path of each input file, and replaced by an optional second element, to obtain the corresponding image id

- **files** : the list of input image files (*file globs* allowed); may be relative to config file location

The image id's of both the input and output files will be collected in an accompanying *<output_name>.meta* file, for later processing by the generator when creating source and build versions of the app. You may move these files around after creation, but you'll have to keep the combined image and its .meta file together in the same directory. At generation time, the generator will look for an accompanying .meta file for every image file it finds in a library. The combined image's image id will be refreshed from its current location relative to the library's resource path. But the clipped images (the images inside the combined image) will be registered under the image id's given in the .meta file (and for browser that don't support combined images, they'll have to be available on disk under this exact image id).

compile

Triggers the generation of a source or build version of the app. Takes a map.

```
"compile" :
{
  "type" : "(source|build)"
}
```

Note: peer-keys: *compile-options*, *cache*, *include*, *library*

Generate Javascript file(s) for the application that can be loaded in the browser. This includes an inital file that acts as a bootstrap/loader file, and possibly other JS files with class code, I18N files, asf. All necessary settings for the compile run are given in the *compile-options* key, so make sure this one is properly filled.

Possible keys are

- **type** : which version of the application should be generated (default: *source*)

compile-options

Specify various options for compile (and other) keys. Takes a map.

```
"compile-options" :
{
  "paths" :
  {
    "file"             : "<path>",
    "app-root"         : "<path>",
    "gzip"             : (true|false),
    "loader-template" : "<path>",
    "scripts-add-hash": (true|false)
  },
  "uris" :
  {
    "script"           : "script",
    "resource"         : "resource",
    "add-nocache-param" : (true|false)
  },
  "code" :
  {
    "format"           : (true|false),
    "locales"          : ["de", "en"],
    "optimize"         : ["variables", "basecalls", "privates", "strings"],
    "decode-uris-plug"  : "<path>"
  }
}
```

Output Javascript file(s) are generated into dirname(<file>), with <file> being the primary file. Within the files, references to other script files are generated using the <script> URI prefix, references to resources will use a <resource> URI prefix. If <file> is not given, the `APPLICATION` macro has to be set in the global *let* section with a proper name, in order to determine a default output file name.

Possible keys are (<type> refers to the *compile/type*, e.g. source or build)

- **paths** : paths for the generated output
  - **file** : the path to the compile output file; can be relative to the config's directory (default: *<type>/script/<appname>.js*)
  - **app-root** : relative (in the above sense) path to the directory containing the app's HTML page (relevant for *source* version; default: *./<type>*)
  - **loader-template** : path to a JS file that will be used as an alternative loader template; for possible macros and structure see the default template in `tool/data/generator/loader.tmpl.js`
  - **gzip** : whether to gzip output file(s) (default: *false*)
  - **scripts-add-hash** : whether the file name of generated script files should contain the script's hash code; the primary compile output file (see above) is exempted even if set to true (default: *false*)
- **uris** : URIs used to reference code and resources
  - **script** : URI from application root to code directory (default: *"script"*)
  - **resource** : URI from application root to resource directory (default: *"resource"*)
  - **add-nocache-param** : whether to add a "?nocache=<random_number>" parameter to the URI, to overrule browser caching when loading the application (relevant for *source* version; default: *true*)
- **code** : code options
  - **format** : whether to apply simple output formatting (it adds some sensible line breaks to the output code) (default: *false*)
  - **locales** : a list of locales to include (default: *["C"]*)
  - **optimize** : list of dimensions for optimization, max. '["variables", "basecalls", "privates", "strings"]' (default: *[]*) *special section*
  - **decode-uris-plug** : path to a file containing JS code, which will be plugged into the loader script, into the `qx.$$loader.decodeUris()` method. This allows you to post-process script URIs, e.g. through pattern matching. The current produced script URI is available and can be modified in the variable `euri`.

copy-files

Triggers files/directories to be copied. Takes a map.

```
"copy-files" :
{
  "files"     : [ "<path>", "<path>" ],
  "source" : "<path>",
  "target"  : "<path>"
}
```

Note: peer-keys: *cache*

Possible keys are

- **files** *(required)* : an array of files/directories to copy; entries will be interpreted relative to the `source` key value

- **source** : root directory to copy from; may be relative to config file location (default: "source")
- **target** : root directory to copy to; may be relative to config file location (default: "build")

copy-resources

Triggers the copying of resources. Takes a map.

```
"copy-resources" :
{
  "target" : "<path>"
}
```

Note: peer-keys: *cache*, *include*, *library*

Possible keys are

- **target** : root target directory to copy resources to; may be relative to the config file location (default: "build")

Unlike *copy-files*, `copy-resources` does not take either a "source" key, nor a "files" key. Rather, a bit of implicit knowledge is applied. Resources will be copied from the involved libraries' `source/resource` directories (this obviates a "source" key). The list of needed resources is derived from the class files (e.g. from `#asset` hints - this obviates the "files" key), and then the libraries are searched for in order. From the first library that provides a certain resource, this resource is copied to the target folder. This way you can use most resources from a standard library (like the qooxdoo framework library), but still "shadow" a few of them by resources of the same path from a different library, just by tweaking the order in which these libraries are listed in the *library* key.

default-job

Default job to be run. Takes a string.

```
"default-job" : "source"
```

If this key is present in a configuration file, the named job will be run by default when no job argument is passed to the generator on the command line.

dependencies

Allows you to influence the way class dependencies are processed by the generator. Takes a map.

```
"dependencies" :
{
  "follow-static-initializers"  : (true|false),
  "sort-topological"            : (true|false)
}
```

- **follow-static-initializers** *(experimental!)*: Try to resolve dependencies introduced in class definitions when calling static methods to initialize map keys (default: *false*).
- **sort-topological** *(experimental!)*: Sort the classes using a topological sorting of the load-time dependency graph (default: *false*).

desc

Provides some descriptive text for the job.

```
"desc" : "Some text."
```

The descriptive string provided here will be used when listing jobs on the command line. (Be aware since this is a normal job key it will be passed on through job inheritance, so when you look at a specific job in the job listing you might see the job description of some ancestor job).

exclude

Exclude classes from processing in the job. Takes an array of class specifiers.

```
"exclude" : ["qx.util.*"]
```

Classes specified through the *exclude* key are excluded from the job processing, e.g. from the generated build output. The class specifiers can include simple wildcards like "qx.util.*" denoting class id's matching this pattern, including sub-name spaces.

A leading '=' in front of a class specifier (like "=qx.util.*") means 'without dependencies'. That means the classes themselves are exempted, but their dependencies added in. Be aware that this requires that *all* dependencies have to be calculated upfront, including for those classes specified in this key, resulting in increased compile time and generator logging. It also means that the final class list might contain classes that are not used by any of the remaining classes. Usually, specifying classes without '=' is what you want.

export

List of jobs to be exported if this config file is included by another, or to the generator if it is an argument.

```
"export" : ["job1", "job2", "job3"]
```

Only exported jobs will be seen by importing config files. If the current configuration file is used as an argument to the generator (either implicitly or explicitly with *-c*), these are the jobs the generator will list with *generate.py x*, and only these jobs will be runnable with *generate.py <jobname>*.

extend

Extend the current job with other jobs. Takes an array of job names.

```
"extend" : [ "job1", "job2", "job3" ]
```

The information of these (previously defined) jobs are merged into the current job description. Keys and their values missing in the current description are added, existing keys take precedence and are retained (with some keys that are merged).

Special section

fix-files

Fix white space in Javascript class files. Takes a map.

```
"fix-files" :
{
  "eol-style" : "(LF|CR|CRLF)",
  "tab-width" : 2
}
```

Note: peer-keys: *library*

fix-files will normalize white space in source code, by converting tabs to spaces, removing trailing white space in lines, and unifying the line end character sequence.

Possible keys are

- **eol-style** : determines which line end character sequence to use (default: *LF*)
- **tab-width** : the number of spaces to replace tabs with (default: *2*)

include

Include classes to be processed in the job. Takes an array of class specifiers.

```
"include" : ["qx.util.*"]
```

The class specifiers can include simple wildcards like 'qx.util.*' denoting a whole set of classes. A leading '=' in front of a class specifier means 'without dependencies' (like '=qx.util.*'). These classes are e.g. included in generated Javascript.

include (top-level)

Include external config files. Takes a list of maps.

```
"include" :
[
  {
    "path"   : "<path>",
    "as"     : "<name>",
    "import" : ["extjob1", "extjob2", "extjob3"],
    "block"  : ["extjob4", "extjob5"]
  }
]
```

Within each specifying map, you can specify

- **path** *(required)*: Path string to the external config file which is interpreted *relative* to the current config file
- **as** : Identifier that will be used to prefix the external job names on import; without it, job names will be imported as they are.
- **import** : List of job names to import; this list will be intersected with the `export` list of the external config, and the resulting list of jobs will be included. : A single entry can also be a map of the form *{"name": <jobname>, "as": <alias>}*, so you can import individual jobs under a different name.
- **block** : List of job names to block during import; this is the opposite of the `import` key and allows you to block certain jobs from being imported (helpful if you want to import most but not all of the jobs offered by the external configuration).

Special section

jobs

Define jobs for the generator. Takes a map.

```
"jobs" :
{
  "<job_name>" : { <job_definition> }
}
```

Job definitions can take a lot of the predefined keys that are listed on this page (see the *overview* to get a comprehensive list). The can hold "actions" (keys that cause the generator to perform some action), or just settings (which makes them purely declarative). The latter case is only useful if those jobs are included by others (through the *extend* key, and thus hold settings that are used by several jobs (thereby saving you from typing).

let

Define macros. Takes a map.

```
"let" :
{
  "<macro_name>"  : "<string>",
  "<macro_name1>" : [ ... ],
  "<macro_name2>" : { ... }
}
```

Each key defines a macro and the value of its expansion. The expansion may contain references to previously defined macros (but no recursive references). References are denoted by enclosing the macro name with `${...}` and can only be used in strings. If the value of the macro is a string, references to it can be embedded in other strings (e.g. like "/home/${user}/profile"); if the value is a structured expression, like an array or map, references to it must fill the entire string (e.g. like "${MyList}").

- <macro_name> : The name of the macro.

Special section

let (top-level)

Define default macros. Takes a map (see the other *'let'*). Everything of the normal 'let' applies here, except that this let map is included automatically into every job run. There is no explicit reference to it, so be aware of side effects.

library

Define libraries to be taken into account for this job. Takes an array of maps.

```
"library" :
[
  {
    "manifest"   : "<path>",
    "uri"        : "<from_html_to_manifest_dir>"
  }
]
```

Each map can contain the keys

- **manifest** *(required)* : path to the "Manifest" file of the library; may be relative to config file location; may use `contrib://` scheme
- **uri** : URI prefix from your HTML file to the directory of the library's "Manifest" file

Special section

lint-check

Check Javscript source code with a lint-like utility. Takes a map.

```
"lint-check" :
{
  "allowed-globals" : [ "qx", "qxsettings", "qxvariants", "${APPLICATION}" ]
}
```

Note: peer-keys: *library*, *include*

Keys are:

- **allowed-globals** : list of names that are not to be reported as bad use of globals

log

Configure log/reporting features. Takes a map.

```
"log" :
{
  "classes-unused" : [ "custom.*", "qx.util.*" ],
  "privates"       : ("on"|"off"),
  "resources"      :
  {
    "file"         : "<filename>"
  }
  "filter"         :
  {
    "debug"        : [ "generator.code.PartBuilder.*" ]
  },
  "dependencies"   :
  {
    "type"         : ("using"|"used-by"),
    "phase"        : ("runtime"|"loadtime")
    "format"       : ("txt"|"dot"|"json"|"provider"|"flare"|"term"),
    "dot"          :
    {
      "root"           : "custom.Application",
      "file"           : "<filename>",
      "radius"         : 5,
      "span-tree-only" : (true|false),
      "compiled-class-size" : (true|false),
      "optimize"       : [<optimize-keys>]
    },
    "json"         :
    {
      "file"       : "<filename>",
      "pretty"     : (true|false)
    },
    "flare"        :
    {
      "file"       : "<filename>",
      "pretty"     : (true|false)
    }
  }
}
```

Note: peer-keys: *cache*, *include*, *library*, *variants*

This key allows you to enable logging features along various axes.

- **classes-unused** : Report unused classes for the name space patterns given in the list.
- **privates** : print out list of classes that use a specific private member
- **resources**: writes the map of resource infos for the involved classes to a json-formatted file
  - **file** : output file path (default *resources.json*)
- **filter** : allows you to define certain log filter
  - **debug** : in debug ("verbose") logging enabled with the `-v` command line switch, only print debug messages from generator modules that match the given pattern
- **dependencies** : print out dependency relations of classes
  - **type** *(required)*: which kind of dependencies to log
    - `using`: dependencies of the current class to other classes; uses the **using** key; supports `txt`, `dot`, `json` and `flare` output formats
    - `used-by`: dependencies of other classes to the current class; supports only `txt` format
  - **phase** : limit logging to runtime or loadtime dependencies (default: *both*)
  - **format** : format of the dependency output (default: *txt*)
    - `txt`: textual output to the console
    - `dot`: generation of a Graphviz dot file; uses the **dot** key
    - `json`: "native" Json data structure (reflecting the hierarchy of the txt output class -> [run|load]); uses the **json** key
    - `provider`: similar to the `json` output, but all id's are given as path suffixes (slashes between name spaces, file extensions), and dependencies are extended with resource id's and translatable string keys (as `translation#<key>`); uses the **json** key
    - `flare`: Json output suitable for Prefuse Flare depencency graphs; uses the **flare** key
    - `term`: textual output to the console, in the form of a term *depends(<class>, [<load-deps>,...], [<run-deps>,...])*
  - **dot**:
    - **span-tree-only**: only create the spanning tree from the root node, rather than the full dependency graph; reduces graph complexity by limiting incoming edges to one (i.e. for all classes at most one arrow pointing to them will be shown), even if more dependency relations exist
    - **root** : the root class for the `dot` format output; only dependencies starting off of this class are included
    - **file** : output file path (default *deps.dot*)
    - **radius** : include only nodes that are within the given radius (or graph distance) to the root node
    - **compiled-class-size** : use compiled class size to highlight graph nodes, rather than source file sizes; if true classes might have to be compiled to determine their compiled size, which could cause the log job to run longer (default *true*)
    - **optimize** : if **compiled-class-size** is true, provide optimization settings here so classes are compiled with the correct optimizations; see *compile-options/code/optimize* for possible values (default [])
  - **json**:
    - **file** : output file path (default *deps.json*)

    - **pretty** : produce formatted Json, with spaces and indentation; if *false* produce compact format (default: *false*)

  - **flare**:

    - **file** : output file path (default *flare.json*)
    - **pretty** : produce formatted Json, with spaces and indentation; if *false* produce compact format (default: *false*)

Special section.

migrate-files

Migrate source files to current qooxdoo version. Takes a map.

```
"migrate-files" :
{
   "from-version" : "0.7",
   "migrate-html" : false
}
```

This key will invoke the mechanical migration tool of qooxdoo, which will run through the class files an apply successive sequences of patches and replacements to them. This allows to apply migration steps automatically to an existing qooxdoo application, to make it better comply with the current SDK version (the version the key is run in). Mind that you might have to do further adaptions by hand after the automatic migration has run. The migration tool itself is interactive and allows entering migration parameters by hand.

- **from-version** : qooxdoo version of the code before migration
- **migrate-html** : whether to patch .html files in the application (e.g. the index.html)

name

Provides some descriptive text for the whole configuration file.

```
"name" : "Some text."
```

packages

Define packages for this app. Takes a map.

```
"packages" :
{
  "parts"  :
  {
    "<part_name>" :
    {
      "include"                  : [ "app.class1", "app.class2", "app.class3.*" ],
      "expected-load-order"      : 1
      "no-merge-private-package" : (true|false)
    }
  },
  "sizes"  :
  {
    "min-package"           : 1,
    "min-package-unshared"  : 1
```

```
  },
  "init"               : "<part_name>",
  "loader-with-boot"  : (true|false),
  "i18n-with-boot"    : (true|false),
  "additional-merge-constraints" : (true|false),
  "verifier-bombs-on-error"      : (true|false)
}
```

Note: peer-keys: *compile*, *library*, *include*

Keys are

- **parts** : map of part names and their properties
  - <part_name> :
    - **include** *(required)*: list of class patterns
    - **expected-load-order** : integer > 0 (default: *undefined*)
    - **no-merge-private-package** : whether the package specific to that individual part should not be merged; this can be used when carving out resource-intensive parts (default: *false*)
- **sizes** : size constraints on packages
  - **min-package** : minimal size of a package in KB (default: 0)
  - **min-package-unshared** : minimal size of an unshared package in KB (default: <min-package>)
- **init** : name of the initial part, i.e. the part to be loaded first (default: *"boot"*)
- **loader-with-boot** : whether loader information should be included with the boot part, or be separate; if set false, the loader package will contain no class code (default: *true*)
- **i18n-with-boot** : whether internationalization information (translations, CLDR data, ...) should be included with the boot part, or be separate; if set false, the loader package will contain no i18n data; rather, i18n data will be generated in dedicated parts, which have to be loaded by the application explicitly; see *special section* (default: *true*)
- **additional-merge-constraints** (*experimental*) : if set to true, will cause additional constraints to be applied when merging packages; might result in more packages per part after part collapsing (default: *false*)
- **verifier-bombs-on-error** (*experimental*) : whether the part verifier should raise an exception, or just warn and continue (default: *true*)

Special section

pretty-print

Triggers code beautification of source class files (in-place-editing). An empty map value triggers default formatting, but further keys can tailor the output.

```
"pretty-print" :
{
  "general" :
  {
    "indent-string"        : "  "
  },
  "comments" :
  {
    "trailing" :
    {
```

```
      "keep-column"         : false,
      "comment-cols"        : [50, 70, 90],
      "padding"             : "  "
    }
  },
  "blocks" :
  {
    "align-with-curlies"    : false,
    "open-curly" :
    {
      "newline-before"      : "m",
      "indent-before"       : false
    }
  }
}
```

Note: peer-keys: *library*, *include*

Keys are:

- **general** : General settings.
  - **indent-string** : "<whitespace_string>", e.g. "t" for tab (default: " " (2spaces))
- **comments** : Settings for pretty-printing comments. * **trailing** : Settings for pretty-printing line-end ("trailing") comments ("%//% ...").
  - **keep-column** : (true|false) Tries to fix the column of the trailing comments to the value in the original source (default: false)
  - **comment-cols** : [n1, n2, ..., nN] Column positions to start trailing comments at, e.g. [50, 70, 90] (default: [])
  - **padding** : "<whitespace_string>" White space to be inserted after statement end and beginning of comment (default: " " (2spaces))
- **blocks** : Settings for pretty-printing code blocks.
  - **align-with-curlies** : (true|false) Whether to put a block at the same column as the surrounding/ending curly bracket (default: false)
  - **open-curly** : Settings for the opening curly brace '{'.
    - **newline-before** : "(a|A|n|N|m|M)" Whether to insert a line break before the opening curly always (aA), never (nN) or mixed (mM) depending on block complexity (default: "m")
    - **indent-before** : (true|false) Whether to indent the opening curly if it is on a new line (default: false)

provider

Collects application classes, resources, translateable strings and dependency information in a specific directory structure, under the `provider` root directory. Takes a map.

```
"provider" :
{
  "app-root" : "./provider",
  "include"  : ["${APPLICATION}.*"],
  "exclude"  : ["${APPLICATION}.test.*"]
}
```

Note: peer-keys: *library*, *cache*

Keys are:

- **app-root** : Chose a different root directory for the output (default: *./provider*).
- **include** : Name spaces for classes and resources to be included (default: *${APPLICATION}.**).
- **exclude** : Name spaces for classes and resources to be excluded (default: *${APPLICATION}.test.**).

require

Define prerequisite classes needed at load time. Takes a map.

```
"require" :
{
  "<class_name>" : [ "qx.util", "qx.fx" ]
}
```

Each key is a

- <class_name> : each value is an array of required classes for this class.

run

Define a list of jobs to run. Takes an array of job names.

```
"run" : [ "<job1>", "<job2>", "<job3>" ]
```

These jobs will all be run in place of the defining job (which is sort of a 'meta-job'). All further settings in the defining job will be inherited by the listed jobs (so be careful of side effects).

Special section

settings

Define qooxdoo settings. Takes a map.

```
"settings" :
{
  "qx.application" : "myapp"
}
```

Possible keys are valid

- <qooxdoo_settings> : along with their desired values

shell

Triggers the execution of external commands. Takes a map.

```
"shell" :
{
  "command" : ("echo foo bar baz"|["echo foo", "echo bar", "echo baz"])
}
```

Note: peer-keys: *cache*

Possible keys are

- **command** : command string or list of command strings to execute by shell

Note: Generally, the command string is passed to the executing shell "as is", with one exception: Relative paths are absolutized, so you can run those jobs from remote directories. In order to achieve this, all strings of the command are searched for path separators (e.g. '/' on Posix systems, '\' on Windows - be sure to encode this as '\\' on Windows as '\' is the Json escape character). Those strings are regarded as paths and - unless they are already absolute - are absolutized, relative to the path of the current config. So e.g. instead of writing

```
"cp file1 file2"
```

you should write

```
"cp ./file1 ./file2"
```

and it will work from everywhere.

simulate

Runs a suite of GUI tests (simulated interaction). Takes a map.

```
"simulate" :
{
  "java-classpath" : ["../rhino/js.jar", "../selenium/selenium-java-client-driver.jar"],
  "qxselenium-path" : "${SIMULATOR_ROOT}/tool",
  "rhino-class" : "org.mozilla.javascript.tools.shell.Main",
  "simulator-script" : "${BUILD_PATH}/script/simulator.js"
}
```

Possible keys are

- **java-classpath** *(required)*: Java classpath argument for Rhino application. Takes an Array. Must point to the Selenium client driver and Rhino JARs. (default: *${SIMULATOR_CLASSPATH}*)
- **qxselenium-path** *(required)*: Location of the QxSelenium Java class. (default: *${SIMULATOR_ROOT}/tool*)
- **rhino-class** *(required)*: Full name of the Mozilla Rhino class that should be used to run the simulation. Set to *org.mozilla.javascript.tools.debugger.Main* to run the test application in Rhino's visual debugger. (default: *org.mozilla.javascript.tools.shell.Main*)
- **simulator-script** *(required)*: Path of the compiled Simulator application to be run. (default: *${ROOT}/simulator/script/simulator.js*)

slice-images

Triggers cutting images into regions. Takes a map.

```
"slice-images" :
{
  "images" :
  {
    "<input_image>" :
    {
        "prefix"       : "<string>",
        "border-width" : (5 | [5, 10, 5, 10]),
        "trim-width"   : (true|false)
    }
  }
}
```

Note: peer-keys: *cache*

- **images** : map with slice entries.
- <input_image> : path to input file for the slicing; may be relative to config file location
- **prefix** *(required)* : file name prefix used for the output files; will be interpreted relative to the input file location (so a plain name will result in output files in the same directory, but you can also navigate away with `../../....` etc.)
- **border-width** : pixel width to cut into original image when slicing borders etc. Takes either a single integer (common border width for all sides) or an array of four integers (top, right, bottom, left).
- **trim-width** : reduce the width of the center slice to no more than 20 pixels. (default: *true*)

translate

(Re-)generate the .po files (usually located in `source/translation`) from source classes. Takes a map. The source classes of the specified name space are scanned for translatable strings. Those strings are extracted and put into map files (.po files), one for each language. Those .po files can then be edited to contain the proper translations of the source strings. For a new locale, a new file will be generated. For existing .po files, re-running the job will add and remove entries as appropriate, but otherwise keep existing translations.

```
"translate" :
{
  "namespaces"                 : [ "qx.util" ],
  "locales"                    : [ "en", "de" ],
  "pofile-with-metadata"       : (true|false)
  "poentry-with-occurrences" : (true|false)
}
```

Note: peer-keys: *cache*, *library*

- **namespaces** *(required)* : List of name spaces for which .po files should be updated.
- **locales** : List of locale identifiers to update.
- **pofile-with-metadata** : Whether meta data is automatically added to a *new* .po file; on existing .po files the meta data is retained (default: *true*)
- **poentry-with-occurrences** : Whether each PO entry is preceded by `#:` comments in the *.po* files, which indicate in which source file(s) and line number(s) this key is used (default: *true*)

use

Define prerequisite classes needed at run time. Takes a map.

```
"use" :
{
  "<class_name>" : [ "qx.util", "qx.fx" ]
}
```

Each key is a

- <class_name> : each value is an array of used classes of this class.

variants

Define variants for the current app. Takes a map.

```
"variants" :
{
  "qx.debug" : [ "on" , "off" ]
}
```

Possible keys are valid

- <variant_key> : (e.g. "qx.debug"), with a list of their desired values (e.g. '["off"]')

As soon as you specify more than one element in the list value for a variant, the generator will generate different builds for each element. If the current job has multiple variants defined, some of them with multiple elements in the value, the generator will generate a variant **for each possible combination** of the given values.

Special section

11.3.3 Configuration Macro Reference

This page lists the macros which are pre-defined in qooxdoo, and can (mostly) be overridden in custom configuration files.

Macro name	Description	Default value
API_EXCLUDE	list of class pattern to exclude from the api documentation	[]
API_INCLUDE	list of class pattern to include in the api documentation	["qx.*", "${APPLICATION}.*"]
APPLICATION	application name space	<undef>
APPLICATION_MAIN_CLASS	application main class	${APPLICATION}.Application
BUILD_PATH	output path for the "build" job (can be rel. to config dir)	./build
CACHE	path to the compile cache (can be rel. to config dir)	${TMPDIR}/cache
HOME	(read-only) value of the (process) environment variable "HOME"	"."
LOCALES	list of locales for this application	["en"]
OPTIMIZE	list of optimization options for build version	["basecalls", "variables", "privates", "strings"]
QOOXDOO_PATH	path to the qooxdoo installation root dir	<undef>
QOOXDOO_VERSION	the current qooxdoo version	1.3.1
QXICONTHEME	icon theme to use for this application	["Tango"]
QXTHEME	theme to use for this application	"qx.theme.Modern"
ROOT	application root dir (rel. to config dir)	"."
SIMULATION_INCLUDE	class pattern to search for GUI test classes	"${APPLICATION}.simulation.*"
SIMULATOR_CLASSPATH	Java classpath argument for GUI test runner	"${SIMULATOR_ROOT}/tool/js.jar: ${SIMULATOR_ROOT}/tool/selenium-java-client-driver.jar"
SIMULATOR_ROOT	path to the framework's simulator component	"${QOOXDOO_PATH}/component/simulator"
TEST_INCLUDE	class pattern to search for unit test classes	"${APPLICATION}.test.*"
TESTS_SCRIPT	script file name for the test application (the "AUT")	"tests.js"
TMPDIR	(read-only) path to tmp directory	(platform-dependent, like /tmp etc.)
USERNAME	(read-only) value of the (process) environment variable "USERNAME"	<undef>

11.4 Glossary

11.4.1 Glossary

API Viewer A popular qooxdoo application, the API Viewer is a class browser for the framework class hierarchy, written in qooxdoo. It allows for customized views, where the framework classes are displayed together with the classes of an application, in order to provide automated application documentation. The data displayed is extracted from the JavaScript source code where it is maintained as JavaDoc-like comments.

Build Process qooxdoo comes with its own build system, usually referred to as the "build process" or "build system". It is a collection of ''make" Makefiles and command line tools. Together they help to maintain a development environment and is seamlessly used throughout the framework, the standard applications that come with qooxdoo, and is recommended for any custom application. Its features encompass checking of dependencies and maintaining lists of used framework classes, generating files to "glue" everything together, copying code, HTML, style and resource files around, pretty-formatting of source code, generating complete and compressed JavaScript files, and creating distribution-ready, self-contained application folders. Particularly, the build system helps to maintain a Source and a Build Version of a qooxdoo application.

Build Version The "Build Version" of a qooxdoo application is the version where all application files together with all relevant framework classes have been compressed and optimized, to provide a self-contained and efficient Web application that can be distributed to any Web environment.

class A JS object created with `qx.Class.define()`.

Compiler TBD

Generator The Generator is the backbone of qooxdoo's Build Process. It is the main tool that drives various other tools to achieve the various goals of the Build Process, like dependency checking, compression and resource management.

Interface An Interface is "a class without implementation", i.e. a class-like structure that only names class features like attributes and methods without providing an implementation. It is created with `qx.Interface.define()`.

Mixin A Mixin is a class you cannot instantiate, but provides a certain set of features. Mixins are the included in "proper" classes to add this feature set without the necessity to re-implement it. It is created with `qx.Mixin.define()`.

Quirks Mode *"Quirks mode refers to a technique used by some web browsers for the sake of maintaining backwards compatibility with web pages designed for older browsers, instead of strictly complying with W3C and IETF standards in standards mode."* [Wikipedia]

RIA Rich internet application. A desktop-like application with menus, toolbars, etc. that runs over the Internet in a browser.

Ribbon *"The ribbon is a graphical user interface widget composed of a strip across the top of the window that exposes all functions the program can perform in a single place, with additional ribbons appearing based on the context of the data."* [Wikipedia]

Window A distinct rectangular region on the screen, usually with borders and a top bar that allows to drag it around. More specifically a browser window.

11.5 License

11.5.1 qooxdoo License

qooxdoo may be used under the terms of either the

- GNU Lesser General Public License (LGPL) http://www.gnu.org/licenses/lgpl.html

or the

- Eclipse Public License (EPL) http://www.eclipse.org/org/documents/epl-v10.php

As a recipient of qooxdoo, you may choose which license to receive the code under. Certain files or entire directories may not be covered by this dual license, but are subject to licenses compatible to both LGPL and EPL. License exceptions are explicitly declared in all relevant files or in a LICENSE file in the relevant directories.

Following are the license text of the two licenses.

GNU Lesser General Public License, version 2.1

author's reputation will not be affected by problems that might be
introduced by others.

Finally, software patents pose a constant threat to the existence of
any free program. We wish to make sure that a company cannot
effectively restrict the users of a free program by obtaining a
restrictive license from a patent holder. Therefore, we insist that
any patent license obtained for a version of the library must be
consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the
ordinary GNU General Public License. This license, the GNU Lesser
General Public License, applies to certain designated libraries, and
is quite different from the ordinary General Public License. We use
this license for certain libraries in order to permit linking those
libraries into non-free programs.

When a program is linked with a library, whether statically or using
a shared library, the combination of the two is legally speaking a
combined work, a derivative of the original library. The ordinary
General Public License therefore permits such linking only if the
entire combination fits its criteria of freedom. The Lesser General
Public License permits more lax criteria for linking other code with
the library.

We call this license the "Lesser" General Public License because it
does Less to protect the user's freedom than the ordinary General
Public License. It also provides other free software developers Less
of an advantage over competing non-free programs. These disadvantages
are the reason we use the ordinary General Public License for many
libraries. However, the Lesser license provides advantages in certain
special circumstances.

For example, on rare occasions, there may be a special need to
encourage the widest possible use of a certain library, so that it becomes
a de-facto standard. To achieve this, non-free programs must be
allowed to use the library. A more frequent case is that a free
library does the same job as widely used non-free libraries. In this
case, there is little to gain by limiting the free library to free
software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free
programs enables a greater number of people to use a large body of
free software. For example, permission to use the GNU C Library in
non-free programs enables many more people to use the whole GNU
operating system, as well as its variant, the GNU/Linux operating
system.

Although the Lesser General Public License is Less protective of the
users' freedom, it does ensure that the user of a program that is
linked with the Library has the freedom and the wherewithal to run
that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1

above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices
stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no
charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a
table of data to be supplied by an application program that uses
the facility, other than as an argument passed when the facility
is invoked, then you must make a good faith effort to ensure that,
in the event an application does not supply such function or
table, the facility still operates, and performs whatever part of
its purpose remains meaningful.

(For example, a function in a library to compute square roots has
a purpose that is entirely well-defined independent of the
application. Therefore, Subsection 2d requires that any
application-supplied function or table used by this function must
be optional: if the application does not supply it, the square
root function must still compute square roots.)
These requirements apply to the modified work as a whole. If
identifiable sections of that work are not derived from the Library,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works. But when you
distribute the same sections as part of a whole which is a work based
on the Library, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote
it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Library.

In addition, mere aggregation of another work not based on the Library
with the Library (or with a work based on the Library) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public
License instead of this License to a given copy of the Library. To do
this, you must alter all the notices that refer to this License, so
that they refer to the ordinary GNU General Public License, version 2,
instead of to this License. (If a newer version than version 2 of the
ordinary GNU General Public License has appeared, then you can specify
that version instead if you wish.) Do not make any other change in
these notices.

Once this change is made in a given copy, it is irreversible for
that copy, so the ordinary GNU General Public License applies to all
subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or
derivative of it, under Section 2) in object code or executable form
under the terms of Sections 1 and 2 above provided that you accompany
it with the complete corresponding machine-readable source code, which
must be distributed under the terms of Sections 1 and 2 above on a
medium customarily used for software interchange.

If distribution of object code is made by offering access to copy
from a designated place, then offering equivalent access to copy the
source code from the same place satisfies the requirement to
distribute the source code, even though third parties are not
compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the
Library, but is designed to work with the Library by being compiled or
linked with it, is called a ”work that uses the Library”. Such a
work, in isolation, is not a derivative work of the Library, and
therefore falls outside the scope of this License.

However, linking a ”work that uses the Library” with the Library
creates an executable that is a derivative of the Library (because it
contains portions of the Library), rather than a ”work that uses the
library”. The executable is therefore covered by this License.
Section 6 states terms for distribution of such executables.

When a ”work that uses the Library” uses material from a header file
that is part of the Library, the object code for the work may be a
derivative work of the Library even though the source code is not.
Whether this is true is especially significant if the work can be
linked without the Library, or if the work is itself a library. The
threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data
structure layouts and accessors, and small macros and small inline
functions (ten lines or less in length), then the use of the object
file is unrestricted, regardless of whether it is legally a derivative
work. (Executables containing this object code plus portions of the
Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may
distribute the object code for the work under the terms of Section 6.
Any executables containing that work also fall under Section 6,
whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or
link a ”work that uses the Library” with the Library to produce a
work containing portions of the Library, and distribute that work
under terms of your choice, provided that the terms permit

modification of the work for the customer's own use and reverse
engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the
Library is used in it and that the Library and its use are covered by
this License. You must supply a copy of this License. If the work
during execution displays copyright notices, you must include the
copyright notice for the Library among them, as well as a reference
directing the user to the copy of this License. Also, you must do one
of these things:

a) Accompany the work with the complete corresponding
machine-readable source code for the Library including whatever
changes were used in the work (which must be distributed under
Sections 1 and 2 above); and, if the work is an executable linked
with the Library, with the complete machine-readable "work that
uses the Library", as object code and/or source code, so that the
user can modify the Library and then relink to produce a modified
executable containing the modified Library. (It is understood
that the user who changes the contents of definitions files in the
Library will not necessarily be able to recompile the application
to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the
Library. A suitable mechanism is one that (1) uses at run time a
copy of the library already present on the user's computer system,
rather than copying library functions into the executable, and (2)
will operate properly with a modified version of the library, if
the user installs one, as long as the modified version is
interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at
least three years, to give the same user the materials
specified in Subsection 6a, above, for a charge no more
than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy
from a designated place, offer equivalent access to copy the above
specified materials from the same place.

e) Verify that the user has already received a copy of these
materials or that you have already sent this user a copy.
For an executable, the required form of the "work that uses the
Library" must include any data and utility programs needed for
reproducing the executable from it. However, as a special exception,
the materials to be distributed need not include anything that is
normally distributed (in either source or binary form) with the major
components (compiler, kernel, and so on) of the operating system on
which the executable runs, unless that component itself accompanies
the executable.

It may happen that this requirement contradicts the license
restrictions of other proprietary libraries that do not normally
accompany the operating system. Such a contradiction means you cannot

use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.
8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to

refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR

OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library 'Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice
That's all there is to it!

Eclipse Public License - v 1.0

Eclipse Public License - v 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:

a) in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and

b) in the case of each subsequent Contributor:

i) changes to the Program, and

ii) additions to the Program;

where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents" mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

2. GRANT OF RIGHTS

a) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.

b) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.

c) Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

d) Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

a) it complies with the terms and conditions of this Agreement; and

b) its license agreement:

i) effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;

ii) effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;

iii) states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and

iv) states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

a) it must be made available under this Agreement; and

b) a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement , including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. The Eclipse Foundation is the initial Agreement Steward. The Eclipse Foundation may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

INDEX