

webix jet

Table of Contents

1. [Introduction](#) 1.1
2. [Deploying App](#) 1.2
3. [Migration from older version](#) 1.3
4. Part I - Basic usage 1.4
 1. [Creating app](#) 1.4.1
 2. [Creating views](#) 1.4.2
 3. [In-app navigation](#) 1.4.3
5. Part II - Webix Jet in details 1.5
 1. [JetApp API](#) 1.5.1
 2. [App Config](#) 1.5.2
 3. [Webpack Configuration](#) 1.5.3
 4. [Views and SubViews](#) 1.5.4
 5. [JetView API](#) 1.5.5
 6. [Navigation](#) 1.5.6
 7. [Popups and Windows](#) 1.5.7
 8. [Async views](#) 1.5.8
 9. [Referencing views](#) 1.5.9
 10. [Models](#) 1.5.10
 11. [Routers](#) 1.5.11
 12. [Events and Methods](#) 1.5.12
 13. [Services](#) 1.5.13
 14. [Tools](#) 1.5.14
 15. [Plugins](#) 1.5.15
 16. [Inner Events and Error Handling](#) 1.5.16
 17. [Jet Recipes](#) 1.5.17

Introduction

Introduction to Webix Jet

Latest update was made on November 18, 2017

This guide provides all the information needed to start creating web applications with Webix Jet. Click [Read](#) to read the full book. The book is also [available on GitHub](#), so you can watch the repo to get the updates.

Webix is a library of UI components and you don't need any special techniques to create apps with it. However, while more and more components are added to a project, there's a risk to get a mess of code. This guide will provide you with an easy and convenient way of creating apps with Webix by means of using Webix Jet framework.



Advantages of Webix Jet

Webix Jet allows you to create flexible, easy maintainable apps, where data and visual presentations are clearly separated, interface elements can be easily combined and reused, all parts can be developed and tested separately - all with minimal code footprint. It has a ready to use solution for all kinds of tasks, from simple admin pages to fully-fledged apps with multiple locales, customizable skins, and user access levels.

Webix Jet is a fully client-side solution and can be used with any REST-based data API. So there aren't any special requirements to the server.

Getting Started

To begin with, you should grab the app package from <https://github.com/webix-hub/jet->

[start/archive/master.zip](#) and unpack it locally. After that, run the following commands in the target folder (this assumes that you have *nodejs* and *npm* installed):

```
npm install
npm start
```

Next, open `http://localhost:8080` in the browser. You'll see the application interface. Let's have a look at what it has inside.

The Application Structure

The codebase of the app consists of:

- the *index.html* file that is a start page and the only html file in the app;
- the *sources/myapp.js* file that creates the app and contains app configuration;
- the *sources/views* folder containing modules for interface elements;
- the *sources/models* folder that includes modules for data operations;
- the *sources/styles* folder for CSS assets;
- the *sources/locales* folder for app locales (you can ignore it for now).

How it Works

The basic principle of creating an app is the following. The app is a single page. We divide it into multiple views, which will be kept in separate files. Thus, the process of controlling the behavior of the app gets much easier and quicker.

In order to navigate between pages, we will change the URL of the page. But as we are writing a single page app, we will change not the main URL, but only the part after the hashbang (#!). The framework will react to the URL change and rebuild the interface from these elements.

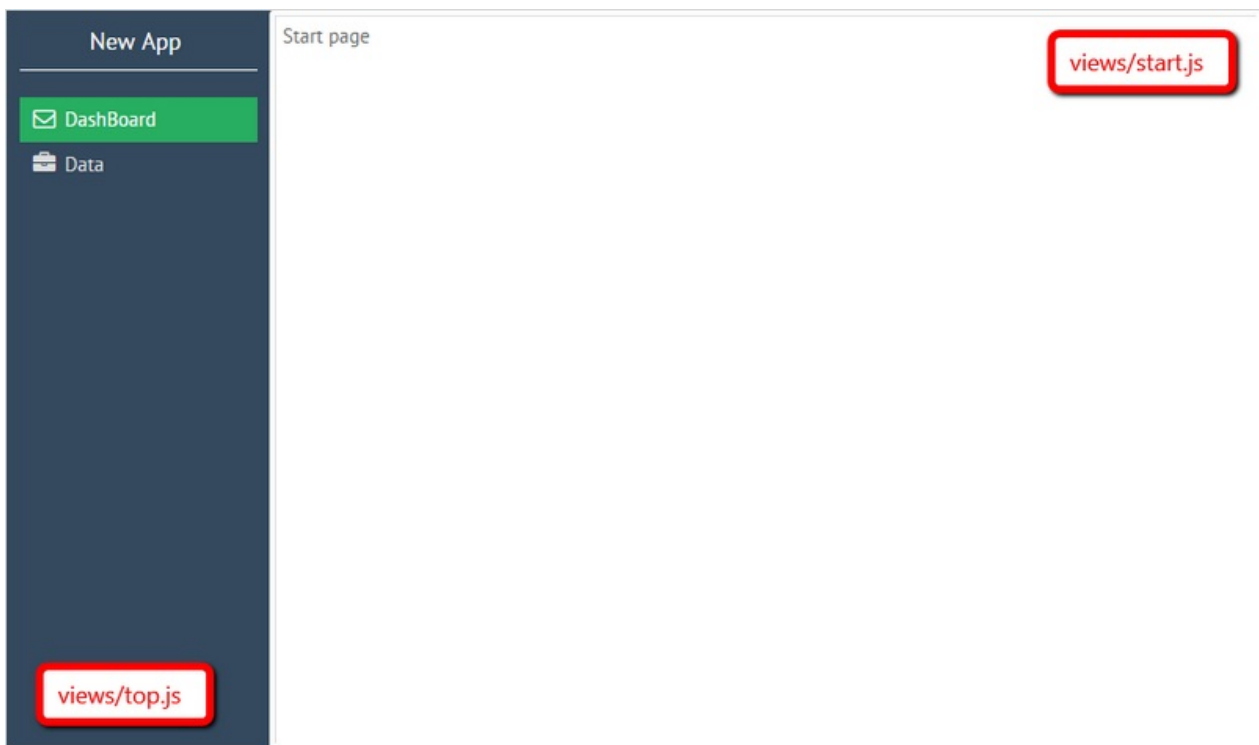
The app splits the URL into parts, finds the corresponding files in the *views* folder and creates an interface by combining UI from those files.

For example, there are 2 files in the *views* folder of our app:

- *top.js*
- *start.js*

If you set the path to *index.html#!/top/start*, the interface described in the *views/top.js* file will be rendered first. Then the interface from *views/start* will be added in some cell of the top-level interface:

`index.html#!/top/start`



Defining a View Module

views/start

The *start.js* file describes a start page view:

```
//views/start.js

export default {
  template:"Start page"
};
```

This is a module that returns a template with the text of the page.

You can look at this page by opening the URL *index.html#!/start*.

views/top

The *views/top* module defines the top level view, that contains a menu and includes the start page view, which we have described above:

```
//views/top.js

import start from "views/start"

export default {
  cols:[
    { view:"menu" },
    start
  ]
};
```

In the above code, there is a layout with two columns. At the top of the file, we are providing the list

of dependencies, which we will use in this layout.

Open the path `index.html#!/top`, and you'll see the page with the *start* view inside of *top*.

Creating Subviews

As it has already been said, our app consists of a single page. How is the process of views manipulation organized?

Check out the following code:

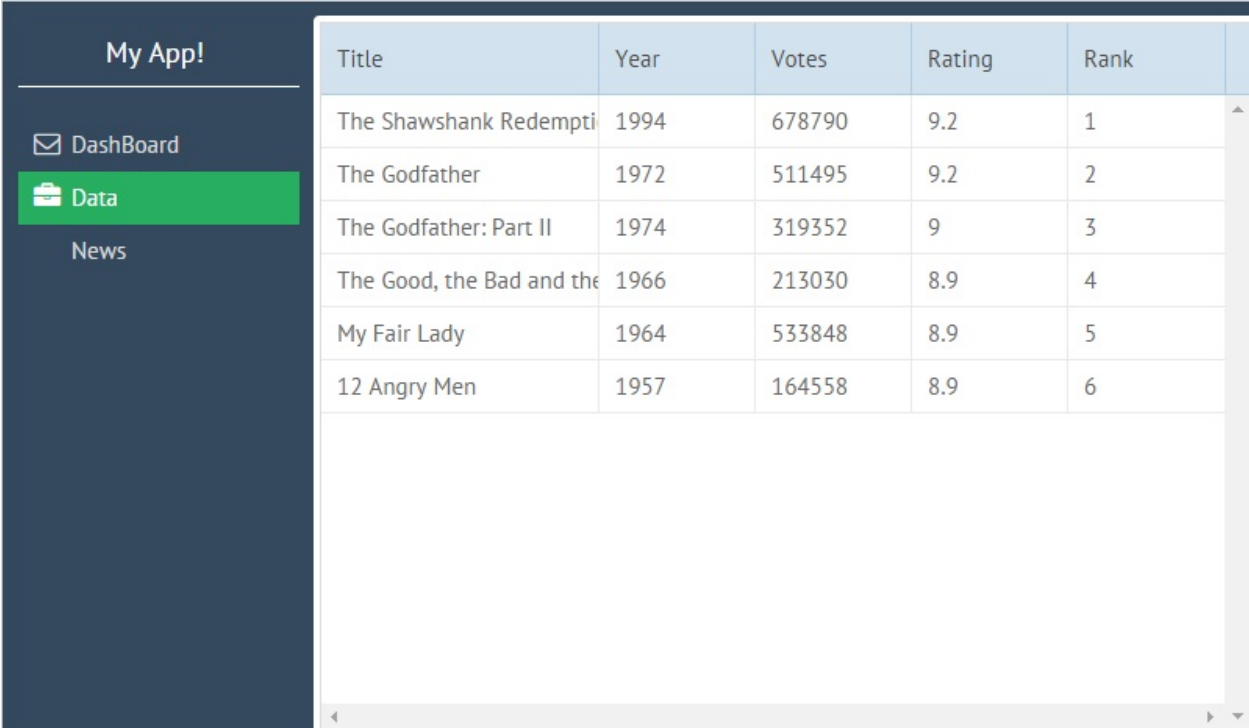
```
//views/top.js

export default {
  cols:[
    { view:"menu" },
    { $subview: true }
  ]
};
```

The line `{ $subview: true }` implies that we can enclose other modules inside of the top module. The next segment of the URL will be loaded into this structure. So for rendering the interface including a particular subview, put its name after `index.html#!/top/` like `index.html#!/top/start`. The `{ $subview: true }` placeholder will be replaced with the content of a subview file (`views/start.js` in the above example) and the corresponding interface will be rendered.

For example, we've got a `data.js` view, which contains a datatable. If you enter the URL `index.html#!/top/data`, you'll get the interface with a menu in the left part and a datatable in the right part:

index.html#!/top/data

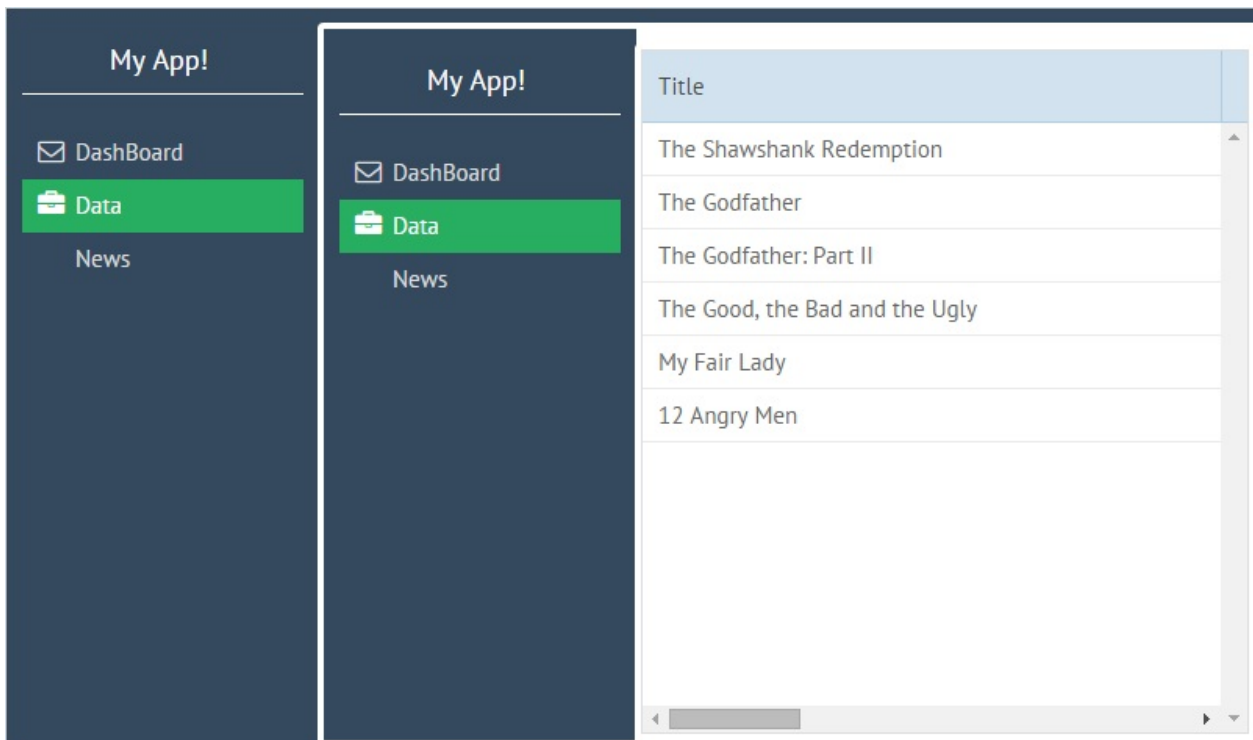


Title	Year	Votes	Rating	Rank
The Shawshank Redempti	1994	678790	9.2	1
The Godfather	1972	511495	9.2	2
The Godfather: Part II	1974	319352	9	3
The Good, the Bad and the	1966	213030	8.9	4
My Fair Lady	1964	533848	8.9	5
12 Angry Men	1957	164558	8.9	6

Then, add one more `/top` subdirectory into the path. The URL will look as `index.html#!/top/top/data`

and the app will have another menu view inserted into the first one. This view will contain the datatable:

index.html#!/top/top/data



The described way of inserting subviews into the main view is an alternative to specifying the necessary subview directly in the main view code.

For more details on including subviews, go to the chapters ["Creating views"](#) and ["Views and Subviews"](#).

Loading Data with Models

While views contain the code of interfaces, models are used to control the data. Let's consider data loading on the example of the `views/data.js` file. It takes data from the `models/records` module. To load data into the view, let's use a data collection. Take a look at the content of the `records.js` file:

```
//models/records.js
export const data = new webix.DataCollection({
  url: "data.php"
});
```

In this module, there is a new data collection that loads data from the `data.php` file. The module returns a helper method that provides access to `DataCollection`.

The `views/data` module has the following code:

```
//views/data.js
import {JetView} from "webix-jet";
import {data} from "models/records";

export default class DataView extends JetView{
  config(){
```

```
        return { view:"datatable", autoConfig:true }  
    }  
    init(view){  
        view.parse(data);  
    }  
};
```

As you can see, this module returns an object that differs from those described earlier. There are two variants of the return object. It can be simply a description of interface or a JetView-based class, which can have:

- the *config* method that returns the interface of the component that will be initialized. In this example, it's a datatable;
- the *init* method that specifies the component initialization behavior, in our case data from the *records* model will be loaded into the view after its creation.

For more details on views and view classes, go to the chapters ["Creating views"](#) and ["Views and Subviews"](#).

For more details on models, [read chapter "Models"](#).

Click to read the full book [Read](#)

Deploying App

Deploying App

When your app is complete and you are ready to let it out in the world, you need to build it for production.

If you are using *webpack.config.js* from the **jet-start** project, you can run either of these commands:

```
npm run build
```

or

```
webpack --env.production true
```

Any of this commands will compile the whole app in two files, **myapp.css** and **myapp.js**, which will be stored in the **codebase** folder (names may differ, based on the webpack config).

Now, you need to upload the files from **codebase** and **index.html** (the html page which is stored at the root of the project) to the production server.

By default, **index.html** uses the CDN version of Webix. If you are using Webix PRO, you need to change paths in **index.html** to the place where *webix.js* and *webix.css* are stored on the production server.

After building an app, you need to include **webix.js** and **codebase/myapp.js**. There are no extra dependencies.

Testing App

To check the quality of the source code, run the following command:

```
npm run lint
```

Migration from older version

Migration Jet 0.x => Jet 1.x

Toolchain and App

- You need to add webpack.config.js and package.json similar to <https://github.com/webix-hub/jet-start>

If you already have package.json in your project, just add the missed dependencies.

```
npm install --save-dev wjet babel-core babel-loader babel-preset-env
npm install webix-jet
```

//or

```
yarn add -D wjet babel-core babel-loader babel-preset-env css-loader
yarn add webix-jet
```

- create folder sources and move all dev files there (app.js, models/, views/, helpers/, etc.)
- in webpack.config.json change entry field to the main file of your app (entry: "sources/app.js")
- update app.js similar to <https://github.com/webix-hub/jet-start/blob/master/sources/myapp.js>
 - import JetApp
 - replace "core.create" with "new JetApp"
 - remove "view.use" commands if any
 - add app.render() call
- run the app

```
npm start
```

- open app at localhost:8080

Migrating views

Jet 1.x can recognize old configuration objects and use them correctly, so you will need not change anything in a common case. There are still some scenarios when you will need to change the code

View has "app" as dependency

Change object to JetView based class and use "this.app" instead of "app"

View is using locale

Change object to JetView and use "this.getService('locale')" instead of 'locale'

Creating app

Creating App

A Webix Jet app is single-page and is divided into views. The application structure consists of the following parts:

- the *index.html* file that is a start page of the app;
- *sources/myapp.js* that contains the configuration of the app, you can give any name to this file;
- the *sources/views* folder that contains elements of the interface;
- the *sources/models* folder that contains data modules.

An app represents an application or an application module. It is used to group views and modules which implement some specific scenario. Later you will be able to combine separate app modules in high-level apps.

The Syntax of Creation

An app module is created as a new instance of the `JetApp` class. You must pass an object with your app configuration like the app name, version, start URL, etc as the parameter.

```
// myapp.js
import {JetApp} from "webix-jet";

var app = new JetApp({
  start:"/top/layout"
}).render(); //mandatory!
```

After you specify the app configuration, you must call the **render** method to build the UI.

You can open the needed URL and the UI will be rendered from the URL elements. The app splits the URL into parts, finds the corresponding files in the **views** folder and creates an interface by combining UI modules from those files.

App Configuration

In the app config, for example, you can set the mode in which the app will work:

```
// myapp.js
import {JetApp} from "webix-jet";

var app = new JetApp({
  mode:"readonly", //application wide configuration
  start:"/top/layout"
});
```

Later in the code, you can do some actions according to the mode:

```
// views/games.js
...
if (this.app.config.mode === "readonly"){
12
```

```
    this.show("limited");  
}  
...
```

this.app refers to the app, while **this** refers to the view¹.

Routers

New Webix Jet has four types of routers. You should specify the preferred router in the app configuration as well. The default router is *HashRouter*. If you don't want to display the hashbang in the URL, you can change the router to *UrlRouter*:

```
// myapp.js  
import {UrlRouter, JetApp} from "webix-jet";  
  
var app = new JetApp({  
    router: UrlRouter,  
    start: "/top/layout"  
});
```

Further reading

You can read these sections of Part II:

- [Routers](#)
- [JetApp API](#)
- [Webpack Configuration](#)

¹↑: You can read more about "[Referencing views](#)" and apps.

Creating views

Creating Views

- [View Concept](#)
- [Static Subviews](#)
- [Dynamic Subviews](#)

View Concept

Views are modules for interface elements. Views should be kept in separate files. This makes the code loosely coupled. If some part of the app is messed up, the rest of it works. Parts of an app can be developed and tested independently. All visual components of the UI are separated from each other and can be reused. All the above said is critical for large and huge apps. Besides, the code looks neater.

All views should be stored in the **views** folder: one view per file. For example, this is how a view module is defined in *views/myview.js*:

```
// views/myview.js
import {JetView} from "webix-jet";

export default class MyView extends JetView{
  config() => { template:"MyView text" };
}
```

You can show the view by opening this path:

`index.html#!/myview`

Subview

Apps created with Webix Jet are single-page. The interface of your app can be constructed from multiple views. Some parts can be dynamic, so you may change them based on the state of the app. Such dynamic views are called *subviews*.

Direct (Static) Including

One of the ways to nest a view is to directly include a view class. Let's create a new view in *bigview.js* and include the **MyView** class into it:

```
// views/bigview.js
import {JetView} from "webix-jet";
import MyView from "myview";

export default class BigView extends JetView{
  config() => {
    rows:[
      MyView,
```

```

        {
            template:"BigView text"
        }
    ]}
}

```

You can open the view with this URL:

```
index.html#!/bigview
```

Dynamic Including

You can enable embedding multiple views that will change according to the URL. Instead of the concrete name of the view class, write `{ $subview: true }`:

```

// views/bigview.js

export default class BigView extends JetView {
    config() => {
        rows:[
            { $subview: true },
            { template:"BigView text" }
        ]}
    }
}

```

The next segment of the URL (after *bigview*) will instruct the app, which view to load as a subview. Based on the URL, the view file will be located, the related view will be loaded, and the corresponding view module will be rendered as a subview.

You can show subviews by changing the URL, for example:

```

//load views/myview.js
index.html#!/bigview/myview

```

```

//load views/viewa.js
index.html#!/bigview/viewa

```

```

//load views/viewb.js
index.html#!/bigview/viewb

```

Further reading

For more information on other ways to define views and include subviews, read:

- [Views and Subviews.](#)

In-app navigation

In-App Navigation (Routing)

Navigation is implemented by changing the URL of the page. The URL reflects the current state of the app. By default, it is stored as a part after the hashbang. Only the part after the hashbang (#!) is changed¹. When the URL is changed, the app updates itself accordingly.

Advantages of Jet Navigation

- *Browser Navigation Keys*: You can move backwards and forwards to previously opened subviews.
- *Refresh Friendly*: If you reload the page, the URL will stay the same and the state of the UI will be exactly the same as before page reload.
- *Convenient Development*: If you work on some particular subview (*games*), you can open the path to it (*#!/games*) and test it separately from the rest of the UI.

In the previous section, "[Creating views](#)", you have read about direct URL navigation. There are three more ways to show views and subviews:

- [Jet links](#)
- [app.show](#)
- [view.show](#)

1. Jet Links

You can add links with the **route** attribute instead of the usual **href** and provide the URL to the desired views, e.g.:

```
<a route="/details/data"></a>
```

After you click on the link, the app UI will be rebuilt and will consist of the parent view *details* and a subview *data*.

You can pass one or more **parameters** with a Jet link:

```
<!-- one -->
<a route="/details/data?id=2"></a>
<!-- or several -->
<a route="/details/data?id=2&name=some"></a>
```

2. app.show()

The **app.show()** method is applied to the whole application and rebuilds its UI. You can call the method from control handlers, for instance.

Here is how you can rebuild the app UI with **app.show**. A specific instance of the related view class is referenced with **this** if you define handlers as *arrow functions*. To reference the app and to call its **show** method, use **this.app**².

```
// views/layout.js
```



```
...
{ view:"button", value:"Details", click: () => {
  this.app.show("/demo/details");
}}
...
```

After a button click, the URL will change, and the app UI will be rebuilt according to it.

App.show with URL Parameters

You can pass one or more parameters to show alongside the URL:

```
// views/layout.js

// one
this.app.show("/demo/details?id=2");
// or many
this.app.show("/demo/details?id=2&name=some");
```

3. view.show()

Rebuilding Part of the App

You can also change the URL by calling the **show()** method of a specific view class. Showing subviews with **view.show** gives you more freedom, as it allows rebuilding only this view or only its subview, not the whole app or app module. For example, suppose you have a view like this:

```
// views/layout.js

export default class LayoutView extends JetView {
  config(){
    return {
      rows:[
        { view:"button", value:"demo" },
        { $subview:true }
      ]
    };
  }
}
```

If the current URL is `/layout/details`, the subview is **details**. Let's replace **details** with the **demo** subview on a button click. To replace the current subview with a different one, pass the name of the subview as it is or with `"/"` to **show**.

A specific instance of the related view class is referenced with **this** if you define a handler as an *arrow function*³. To rebuild a part of the UI, call **this.show()**:

```
// views/layout.js

export default class LayoutView extends JetView {
  config(){
    return {
      rows:[
        { view:"button", value:"demo", click: () => {
```

```

        this.show("demo");
    }},
    { $subview:true }
  ]
};
}
}

//or
...
{ view:"button", value:"demo", click: () => {
    this.show("./demo");
  }}
...

```

If you click **demo**, the resulting URL is going to be `/layout/demo`.

Rebuilding the whole app

If you want to rebuild the whole app and load the **demo** view as the only view, specify the name of the view with a slash:

```

// views/layout.js
...
{ view:"button", value:"demo", click: () => {
    this.show("/demo");
  }}
...

```

View.show with URL Parameters

You can pass one or more parameters with the URL:

```

// views/layout.js

this.show("demo?id=2");
// or many
this.show("demo?id=2&name=some");

```

Further reading

This is all about Webix Jet in a nutshell.

You can also read these sections of Part II:

- [Navigation](#)
- [JetApp API](#)
- [JetView API](#)

[1↑](#): This is relevant for *HashRouter*, which is the default router. Hashbang is not displayed if you use *UrlRouter*. The app part of the URL isn't displayed at all if you use other types of routers. However, the app URL is stored for all the three routers except *EmptyRouter* and the behavior is the same as if the URL were displayed. For more details, [see section "Routers"](#).

[2 ↑](#), [3 ↑](#): To read more about how to reference apps and view classes, go to ["Referencing views"](#).

JetApp API

JetApp API

Here you can find the list of all the **JetApp** methods, that you can make use of.

Method	Use to
attachEvent(event, handler)	attach an event
callEvent(event)	call an event
getService(name, handler)	access a service by its name
render(container)	render the app or the app module
setService(name)	set a service
show(url)	rebuild the app or app module according to the new URL
use(plugin, config)	switch on a plugin

app.attachEvent("event:name", handler)

Use this method to attach a custom event:

```
// views/form.js
import {JetView} from "webix-jet";

export default class FormView extends JetView{
  init(){
    this.app.attachEvent("save:form", function(){
      this.show("aftersave");
    });
  }
}
```

or to attach in inner Jet event:

```
// myapp.js
...
app.attachEvent("app:guard", function(url, view, nav){
  if (url.indexOf("/blocked") !== -1)
    nav.redirect="/somewhere/else";
});
...
```

Events can be attached both in the app file and in view modules.

For more details on events, read ["Events and Methods"](#) and ["Inner Events and Error Handling"](#).

app.callEvent("event:name")

Use this method to call a custom event:

```
// views/data.js
```

```
import {JetView} from "webix-jet";

export default class DataView extends JetView{
  config(){
    return {
      view:"button", click:() => {
        this.app.callEvent("save:form");
      }
    }
  }
}
```

Normally, inner events are called automatically, so there is no need to use **callEvent** for them.

For more details on events, read ["Events and Methods"](#) and ["Inner Events and Error Handling"](#).

app.getService(name)

The method returns a service by its name, passed to the method as a parameter. Call this method to use a service:

```
// views/form.js
import {JetView} from "webix-jet";
import {getData} from "../models/records";

export default class FormView extends JetView{
  config(){
    return {
      view:"form", elements:[
        { view:"text", name:"name" }
      ]
    };
  }
  init(){
    var id = this.app.getService("masterTree").getSelected();
    this.getRoot().setValues(getData(id));
  }
}
```

You can read more about services in the ["Services"](#) chapter.

app.render()

The **render** method builds the UI of the application. If called without any parameters, it just renders the UI inside the page according to the start URL, specified in the app configuration.

```
// myapp.js
...
app.render();
```

But if you want to render the app inside a container, you can pass the string parameter to it with the ID of the container:

```
// myapp.js
```

```
...
app.render("mybox");
```

app.setService(name,handler)

The method initializes a service for view communication.

```
// views/tree.js
import {JetView} from "webix-jet";

export default class treeView extends JetView{
  config(){
    return { view:"tree" };
  }
  init() {
    this.app.setService("masterTree", {
      getSelected : () => this.getRoot().getSelectedId();
    })
  }
}
```

this refers to the instance of the *treeView* class if it is used in an *arrow function*¹.

You can read more about services in the ["Services"](#) chapter.

app.show(url)

The **show** method is used to change the app interface. This method rebuilds the whole UI of the app according to the URL passed as a parameter:

```
// views/some.js
...
app.show("/demo/details")
```

For more info about showing UI components, visit the ["Navigation"](#) chapter.

app.use(plugin, config)

The **use** method is used to switch on plugins. The method takes two parameters:

- the name of the plugin
- the plugin configuration

```
// myapp.js
import session from "models/session";
...
app.use(plugins.User, { model: session });
```

For more details, go to the ["Plugins"](#) chapter.

¹↑: To read more about how to reference apps and view classes, go to ["Referencing views"](#).

App Config

App Configuration

An app module is created as a new instance of the JetApp class. You must pass an object with your app configuration to the app constructor. You can include various parameters into the app configuration.

Parameter	What For
start	to set the start app URL
debug	to enable debugging
router	to change a router
arbitrary parameters	e.g. access mode, screen size, etc
views	to change view modules names
routes	to shorten the app URL

Start URL

The app UI will be rendered from the URL elements from **start** when you open the app for the first time. The app splits the URL into parts, finds the corresponding files in the **views** folder and creates an interface by combining UI modules from those files.

```
// myapp.js
import {JetApp} from "webix-jet";

const app = new JetApp({
  start: "/top/layout"
}).render();
```

Debugging

You can enable debugging in the app configuration:

```
// myapp.js
import {JetApp} from "webix-jet";

const app = new JetApp({
  start: "/top/about",
  debug: true
}).render();
```

With *debug:true*, error messages will be logged into console and a debugger will pause the app on errors.

Various App Parameters

In the app config, for example, you can set the mode in which the app will work:

```
// myapp.js
```



```
import {JetApp} from "webix-jet";

const app = new JetApp({
  mode:"readonly", //application wide configuration
  start:"/top/layout"
}).render();
```

Later in the code, you can do some actions according to the mode:

```
// views/games.js
...
if (this.app.config.mode === "readonly"){
  this.show("limited");
}
...
```

this.app refers to the app, while **this** refers to the view¹.

Routers

New Webix Jet has four types of routers. You should specify the preferred router in the app configuration as well. The default router is *HashRouter*. If you don't want to display the hashbang in the URL, you can change the router to *UrlRouter*:

```
// myapp.js
import {UrlRouter, JetApp} from "webix-jet";

const app = new JetApp({
  router: UrlRouter,
  start:"/top/layout"
}).render();
```

Changing View Names

If the module you want to show is in a subfolder and you want to show the module with a shorter name, you can change the view name in app configuration:

```
// myapp.js
import {JetApp} from "webix-jet";

const app = new JetApp({
  start: "/top/start",
  views: {
    "start" : "area.list" // load /views/area/list.js
  }
}).render();
```

In this example, **list** module is stored in the **area** subfolder in */views* (*/views/area/list.js*). Later, you can show the view by the new name, e.g.:

```
// views/top
import {JetView} from "webix-jet";

export default class TopView extends JetView {
```

```

    config(){
      return {
        cols:[
          { view:"button", value:"start",
            click:(id) => {
              this.show("start");
            }},
          { $subview: true }
        ]
      };
    }
  }
}

```

this in the button handler refers to the Jet view, because the handler is an arrow function¹.

[Check out the demo >>](#)

Beautifying the URL

If you do not want to display some part of the app URL, you can hide it with the help of **routes** in app configuration. For instance, you might want to display only the names of subviews in the URL:

```

// myapp.js
import {JetView} from "webix-jet";

const app = new JetApp({
  start: "/top/about",
  routes: {
    "/hi"      : "/top/about",
    "/form"    : "/top/area.left.form",
    "/list"    : "/top/area.list",
  }
});

```

Instead of a long URL with subdirectories, e.g. *"/top/area.left.form"*, the app URL will be displayed as */form*.

[Check out the demo >>](#)

¹↑: To read more about how to reference apps and view classes, go to ["Referencing views"](#).

Webpack Configuration

Webpack Config

There are some cases when you might want to change the default webpack configuration.

- [Multiple Start Files](#)
- [Turning Off Localization and Views](#)

Multiple Start Files

By default, the app is built with one start file:

```
/* webpack.config.js */
...
var config = {
  entry: "./sources/admin.js",
  output: {
    //...
    filename: "admin.js"
  },
  ...
}
```

To create multiple entry files, pass an object to **entry** and use the *[name]* substitution for output filenames to ensure that each file has a unique name:

```
/* webpack.config.js */
{
  entry: {
    admin: "./sources/admin.js",
    orders: "./sources/orders.js"
  },
  output: {
    //...
    filename: "[name].js"
  }
}
```

Turning Off Localization and Views

If you aren't planning to localize the app, there's a way to do it without creating an empty folder for locales. Without changes in webpack config, you would have to do that to get the app compiled. webpack config has the **resolve** property that presents options affecting the resolving of modules. It tells webpack where to look for locales or some other files, e.g. views. Change the path in the "jet-locales" key pair from **resolve.alias**.

```
/* webpack.config.js */
var config = {
```

```
...
resolve: {
  extensions: [".js"],
  modules: ["/sources", "node_modules"],
  alias:{
    "jet-views":path.resolve(__dirname, "sources/views"),
    "jet-locales":path.resolve(__dirname, "sources/locales")
  }
},
...
}
```

Besides, if you do not want webpack to look for views in the **sources/views** folder, modify the "jet-views" key pair as well. And you can change the "/sources" directory for your modules as well.

Views and SubViews

Views and SubViews

- Views
 - [Simple Views](#)
 - [Object "Factory Pattern"](#)
 - [Class Views](#)
- [SubView Including](#)
 - [View Inclusion](#)
 - [App Inclusion](#)

After reading the "Basics" chapter of this guide, you are familiar with the concept of a *view*. Now it's time to find out all the ways of creating views. You can create views in three ways.

1. Simple Views

Views can be created as pure objects.

Advantage

- This is a simple way to create a view.

Disadvantages

- Simple views are static and are included as they are.
- Simple views have no **init** and other methods that classes have.

Here's a simple view with a list:

```
/* views/list.js */
export default {
  view:"list"
}
```

or

```
const list = {
  view:"list"
};
export default list;
```

2. Object "Factory Pattern"

View objects can also be returned by a factory function.

Advantages

- Such views are still simple.
- Such views are dynamic.

Disadvantages

- Such views have no **init** or other methods that classes have.

Here's a list view returned by a factory:

```
/* views/details.js */
export default () => {
  var data = [];
  for (var i=0; i<10; i++) data.push({ value:i });

  return {
    view:"list", options:data
  }
}
```

3. Class Views

Views can be defined as JS6 classes.

Advantages of Classes

- Views defined as classes are **dynamic** and each new instance can be changed when it's created.
- View classes have **init** and other **methods** that can be redefined by users.
- You can also define **custom methods** and **local variables**.
- All instances have their individual **inner states**. E.g. if you use the same Toolbar class to add identical toolbars at the top and at the bottom, there are two instances of a Toolbar class and the toolbars will behave independently.
- Classes have the **this** pointer that references the view inside methods and handlers.
- You can **extend** class views. With JS6 classes, inheritance is closer to classic OOP and the syntax is nicer. Inheritance can help you reuse old components for creating slightly different ones. For example, if you already have a toolbar and want to create a similar one, but with one additional button, define a new class and inherit from the old toolbar.

JetView Methods

View classes inherit from **JetView**. Webix UI lifetime event handlers are implemented through class methods. Here are the methods that you can redefine while defining your class views:

- [config\(\)](#)
- [init\(\)](#)
- [urlChange\(\)](#)
- [ready\(\)](#)
- [destroy\(\)](#)

config()

This method returns the initial UI configuration of a view. Have a look at a toolbar:

```
// views/toolbar.js
import {JetView} from "webix-jet";

export default class ToolbarView extends JetView{
  config(){
    return {
      view:"toolbar", elements:[
        { view:"label", label:"Demo" },
        { view:"segmented", options:["details", "dash"]} ]
    };
  }
}
```

config of *ToolbarView* class returns a simple Webix toolbar.

init(view, url)

The method is called only once for every instance of a view class when the view is rendered. It can be used to change the initial UI configuration of a view returned by **config**. For instance, the above-defined toolbar will be always rendered with the first segment of the button active. You can change the control state in **init**. Let's link it to the URL:

```
// views/toolbar.js
import {JetView} from "webix-jet";

export default class ToolbarView extends JetView{
  config(){
    return {
      view:"toolbar", elements:[
        { view:"label", label:"Demo" },
        { view:"segmented", options:["details", "dash"]} ]
    };
  }
  init(view, url){
    if (url.length > 1)
      view.queryView({view:"segmented"}).setValue(url[1].page)
  }
}
```

The **init** method receives two **parameters**:

1. view - the view UI

The segmented button is referenced by **view.queryView()**. **view** is received by **init** as one of the two parameters and references a Webix view inside the class instance. **queryView** looks for a view (a segmented button in this case) by its attributes. For more details on referencing nested views, [read the "Referencing views" section](#).

2. url - the app URL as an array

Each array element is an object that contains:

- **page** - the name of the URL element
- **params** - parameters that you can pass with the URL
- **index** - the index of the URL element (beginning from 1)

So when **setValue** in the code above was passed `url[1].page`, it received the name of the current subview (*details* or *dash*).

`url.length > 1` checks that a subview is present in the URL.

urlChange(view,url)

This method is called every time the URL is changed. It reacts to the change in the URL after `!#`. **urlChange** is only called for the view that is rendered and for its parent. Consider the following example. The initial URL is:

`/layout/demo/details`

If you change it to:

`/layout/demo/preview`

urlChange will be called for **preview** and **demo**.

The **urlChange** method can be used to restore the state of the view according to the URL, e.g. to highlight the right controls.

Let's expand the previous example with a toolbar and add a click handler to the segmented button that will change the URL:

```
// views/toolbar.js
import {JetView} from "webix-jet";

export default class ToolbarView extends JetView{
  config(){
    return {
      view:"toolbar", elements:[
        { view:"label", label:"Demo" },
        { view:"segmented", options:["details", "dash"], click:
          this.$scope.show(this.getValue());
        }
      ]
    };
  }
}
```

As the click handler is a simple function, you must refer to the Jet view class instance as **this.\$scope** to call its **show** method. **this** in simple *function* handler refers to the Webix control, the segmented button in this case. You can read more on ["Referencing views"](#).

Here's how you can select the right segment of the button in **urlChange**:

```
// views/toolbar.js
urlChange(view, url){
  if (url.length > 1)
```



```

        view.queryView({view: "segmented"}).setValue(url[1].page)
    }

```

The **urlChange** method receives the same two **parameters** as **init**:

- **view** - the Webix view inside the Jet view class
- **url** - the URL as an array of URL elements

ready(view,url)

ready is called when the current view and all its subviews have been rendered. For instance, if the URL is changed to *a/b*, the order in which view class methods are called is the following:

```

config a
init a
    config b
    init b
    urlChange b
    ready b
urlChange a
ready a

```

Here's how you can use **ready**. There are two simple views, a list and a form for editing the list:

```

// views/list.js
const list = {
    view: "list",
    select: true,
    template: "#value#",
    data: [{value: "one"}, {value: "two"}]
};

// views/form.js
const form = {
    view: "form",
    elements: [
        {view: "text", name: "value", label: "Value"},
        {view: "button", value: "Save", width: 90}
    ]
};

```

Let's include these views into one module and bind the list to the form:

```

// views/listedit.js
import {JetView} from "webix-jet";
import list from "list";
import form from "form";

export default class ListEditView extends JetView{
    config(){
        return {
            cols:[
                { $subview: list, name: "list" },
                { $subview: form, name: "form" }
            ]
        }
    }
}

```

```

        ]
    }
}
ready(){
    this.getSubview("form").bind(this.getSubview("list"));
}
}

```

In the example, the form will be bound to the list only when both the list and the form are rendered.

Note that *subviews* can have **names**. If you give a name to a subview, you can reference it as **this.getSubview("name")**, where **this** is the instance of the class view that includes this subview. You can read more on ["Referencing views"](#).

ready receives same two **parameters**:

- **view** - the Webix view inside the Jet view class
- **url** - the URL as an array of URL elements

destroy()

destroy is called only once for each class instance when the view is destroyed. The view is destroyed when the corresponding URL element is no longer present in the URL.

```

// views/toolbar.js
import {JetView} from "webix-jet";

export default class ToolbarView extends JetView{
    config(){
        return {
            view:"toolbar", elements:[
                { view:"label", label:"Demo" },
                { view:"segmented", options:["details", "dash"], click:
                    this.$scope.show(this.getValue())
                }
            ]
        };
    }
    destroy(){
        webix.message("I'm dying!");
    }
}

```

This is all on view class methods.

Which Way to Define Views is Better

If you still doubt which way to choose for defining views, here's a summary.

All ways provide nearly the same result.

When you are using the "**class**" approach, you can define UI config and *init|urlChange|ready|destroy* handlers.

When you are using the "**factory function**" approach, you can define a dynamic UI config without lifetime handlers.

When you are defining views as **const** (*simple view objects*), you can define UI config only.

So if you are choosing between **classes** and **const**, it is flexibility VS brevity.

If you are not sure which one to use, use classes. A class with the **config** method works exactly the same as the **const** declaration.

Subview Including

Apart from direct inclusion [described in the second chapter](#), there are two more ways of including subviews. Let's recap all the possible ways in short:

1. Direct Static Including

- plain including:

```
import child from "child";
...
{
  rows:[
    { view:"button" },
    child
  ]
}
```

- one view including with \$Subview:view:

```
import child from "child";
...
{
  rows:[
    { view:"button" },
    { $Subview:child }
  ]
}
```

- including a hierarchy of views with \$Subview:"top/some":

```
import child from "child";
import grandchild from "grandchild";
...
{
  rows:[
    { view:"button" },
    { $Subview:"child/grandchild" }
  ]
}
```

2. Dynamic Including

- { \$subview:true }

Subview Inclusion in Details

You can include views and apps into other views.

1. View Inclusion

For example, here are three views created in different ways:

- a class view

```
// views/myview.js
```

```
export default class MyView extends JetView {
  config() => { template:"MyView text" };
}
```

- a simple view object

```
// views/details.js
```

```
export default Details = {
  cols: [
    { template:"Details text" },
    { $subview:true }
  ]
}
```

- a view returned by a factory

```
// views/form.js
```

```
export default Form = () => {
  view:"form", elements:[
    { view:"text", name:"email", required:true, label:"Email" },
    { view:"button", value:"save", click:() => this.show("detail:
  ]
}
```

{ \$subview:true } is a placeholder for a dynamically included subview.

Let's group them into a bigger view:

```
// views/bigview.js
```

```
import myview from "myview";
import details from "details";
import form from "form";

export default BigView = {
  rows:[
    myview,
    { $subview:"/details/form" }
  ]
}
```

```
}
```

Mind that all these views could be put in any order you want and it doesn't depend whether they are classes or objects, e.g.:

```
// views/bigview.js

export default BigView = {
  rows:[
    details,
    { $subview:"/myview/form" }
  ]
}
```

View Inclusion into popups and Windows

You can also include a view into a **popup** or a **window**:

```
// views/some.js
...
init(){
  this.win1 = this.ui(WindowsView);
  //this.win1.show();
}
```

where *WindowsView* is a view class like the following:

```
// views/window.js
import {JetView} from "webix-jet";

export default class WindowsView extends JetView{
  config(){
    return { view:"window", body:{} };
  }
  show(target){
    this.getRoot().show(target);
  }
}
```

For more details about popups and windows, [go to the "Popups and Windows" section](#).

2. App Inclusion

App is a part of the whole application that implements some scenario and is quite independent. It can be a subview as well. By including apps into other apps, you can create high-level applications. E.g. here are two views:

```
// views/form.js
import {JetView} from "webix-jet";

export default class FormView extends JetView {
  config() {
    return {
      view: "form",
```

```

        elements: [
            { view: "text", name: "email", required: true, label: "Email" },
            { view: "button", value: "save", click: () => this.save() }
        ]
    };
}
}

// view/details.js
export default DetailsView = () => ({
    template: "Data saved"
});

```

Let's group these views into an app module:

```

// views/app1.js
import {JetApp, EmptyRouter} from "webix-jet";

export var app1 = new JetApp({
    start: "/form",
    router: EmptyRouter //!
}); //no render!

```

Note that this app module isn't rendered. The second important thing is the choice of the router. As this is the inner level, it can't have URL of its own. That's why *EmptyRouter* is chosen. [Go to the "Routers" section](#) for details.

Next, the app module is included into another view:

```

// views/page.js
import {app1} from "app1";
import {toolbar} from "toolbar";

export default PageView = () => ({
    rows: [ toolbar, app1 ]
});

```

Finally, the view can also be put into another app:

```

// app2.js
import {JetApp, HashRouter} from "webix-jet";

var app2 = new JetApp({
    start: "/page",
    router: HashRouter
}).render();

```

As a result, this is a two-level app. [Check out the demo](#).

1: This is true if you use *HashRouter*. There's no hashbang with other routers, but this still works for *URL* and *Store* routers. The URL isn't stored only for *EmptyRouter*. For details, [go to the "Routers" section](#).

JetView API

JetView API

JetView class has the following methods:

Method	Use it to
<code>this.use(plugin, config)</code>	switch on a plugin
<code>this.show("path")</code>	show a view or a subview
<code>this.ui(view)</code>	create a popup or a window
<code>this.on(app,"event:name",handler)</code>	attach an event
<code>this.getRoot()</code>	call methods of the Webix view
<code>this.getSubview(name)</code>	call the methods of a subview
<code>this.getParentView()</code>	call methods of a parent view
<code>this.\$("controlID")</code>	call methods of some Webix view

this.use(plugin, config)

The method includes a plugin, for example, this is how the **Status** plugin can be added:

```
init(){
    this.use(plugins.Status, {
        target: "app:status",
        ajax:true,
        expire: 5000
    });
}
```

For more details on plugins, check out the ["Plugins" section](#).

this.show("path")

This method is used to reload view modules according to the path specified as a parameter:

```
/* sources/views/toolbar.js */
const Toolbar = {
    view: "toolbar",
    elements: [
        { view: "label", label: "Demo" },
        { view: "button", value:"Details",
            click: () => {
                this.show(this.getValue().toLowerCase());
            }
        }
    ]
};
export default Toolbar;
```

For more details on view navigation, [read the "Navigation" article](#).

this.ui(view)

this.ui call is equivalent to **webix.ui**. It creates a new instance of the view passed to it as a parameter. For example, you can create views inside popups or modal windows with **this.ui**. The good thing about this way is that it correctly destroys the window or popup when its parent view is destroyed.

For example, you want to create a view with a list of orders and a form for editing records in the list. The form will be created inside a modal window. Here's the window:

```
// views/orderform.js
const orderform = {
  view: "window",
  id: "order-win",
  modal: true,
  head: "Add new order",
  body: {
    view: "form", id: "order-form", elements: [
      { view: "combo", name: "product", label: "Product", id: "order-product",
        options: [
          { id: 1, value: "Webix Chai"}, { id: 2, value: "Webix Sy"},
        ]},
      { view: "button", label: "Add", type: "form", align: "center",
        webix.$$("order-win").hide();
      }
    ]
  }
};
export default orderform;
```

Have a look at the parent view with a list of records:

```
// views/orders.js
import data from "orders";
import orderform from "orderform";
import {JetView} from "webix-jet";

export default class OrdersView extends JetView{
  config(){
    return {
      rows: [
        { view: "button", type: "iconButton", icon: "plus",
          click: function(){
            this.$scope._form.show(this.$view);
          }},
        { view: "datatable" }
      ]
    }
  }
  init(view){
    view.queryView({ view: "datatable" }).parse(data);
    this._form = this.ui(orderform);
  }
}
```

The window is created in **init** of OrdersView and shown on a button click. And note again that there's no need to destroy the window manually.

For more details about popups and windows, [go to the "popups and Windows" section](#).

this.on(app,"app:event:name",handler)

Use this method to attach events. This way of attaching an event is convenient, because it automatically detaches the event when the view that called it is destroyed.

```
export default class FormView extends JetView{
  init(){
    this.on(this.app, "save:form", function(){
      this.show("aftersave");
    });
  }
}
```

For more details on attaching and calling events, read the ["Events and Methods" section](#).

this.getRoot()

Use this method to return the Webix view inside a Jet class view and to call Webix view methods.

```
// views/form.js
export default class FormView extends JetView{
  config(){
    return {
      view:"form", elements:[
        { view:"text", name:"email", required:true, label:"Email"},
        { view:"button", value:"save",
          click: () => {
            if (this.getRoot().validate())
              this.show("details");
          }
        }
      ]};
  }
}
```

For more details on referencing views, [read the "Referencing" section](#).

this.getSubview(name)

Use this method if you want to get to the methods of a subview. It looks for a subview by its name. So you must set the name. You can do it like this:

```
// views/listedit.js

import {JetView} from "webix-jet";
import ChildList from "list";
import ChildForm from "form";

export default class ListEditView extends JetView{
  config(){
```

```

        return {
            cols:[
                { $subview:list, name:"list" },
                { $subview:form, name:"form" }
            ]
        }
    }
}

```

After you set the name to a subview, you can refer to it with **this.getSubView(name)** from the methods of the parent:

```
// views/listedit.js
```

```

import {JetView} from "webix-jet";
import ChildList from "list";
import ChildForm from "form";

export default class ListEditView extends JetView{
    ...
    ready(){
        var list = this.getSubview("list").getRoot();
        this.getSubview("form").bind(list);
    }
}

```

For more details on referencing views, [read the "Referencing" section](#).

this.getParentView()

Use this method to get to the methods of the parent view.

```

// views/form.js
export default class Child extends JetView{
    config(){
        return {
            view:"form", elements:[
                { view:"text", name:"name" }
            ]
        };
    }
    init(view){
        var item = this.getParentView().getSelected();
        view.setValues(item);
    }
}

```

The child refers to its parent view with **this.getParentView** and calls its **getSelected** method.

For more details, [read the "Referencing" section](#).

See details in ["Events and Methods"](#).

this.\$\$("controlID")

Use **this.\$\$** to look for nested views by their IDs.

```
// views/toolbar.js
export default class ToolbarView extends JetView {
  config() {
    //...
    { view: "segmented", localId: "control", options: ["details",
      click: function() {
        this.$scope.app.show("/demo/" + this.getValue());
      }
    }
  }
  init(ui, url) {
    if (url.length > 1)
      this.$$("control").setValue(url[1].page);
  }
}
```

For details, [read the "Referencing" section](#).

Navigation

Navigation

There are several ways to navigate through an app with Webix Jet:

- [Jet links](#)
- [HTML links](#)
- [app.show](#)
- [view.show](#)

1. Jet Links

Jet links are created with the **route** attribute instead of the usual **href**. The **route** attribute allows you to stop users from leaving the current view. This can be useful in the case of unsaved data, for instance. See the details in the [section on plugins](#). Here's an example of a Jet link:

```
<a route="/details/data"></a>
```

After you click on the link, the app UI will be rebuilt and will consist of the main view *details* and a subview *data*. Note that there is no hashbang in the path.

Route for Webix Controls

The **route** attribute can be added not only to links, but also to Webix controls and widgets. **route** refers to the path to the module inside the **views** folder. For instance, if you want to load the *list* module in the *area* subfolder, you can add **route** to a button with *area.list*:

```
// views/top
import {JetView} from "webix-jet";

export default class TopView extends JetView {
  config(){
    return {
      cols:[
        { view:"button", value:"List", route:"area.list",
          click:function(id){
            var button = this;
            this.$scope.show(button.route);
          }},
        { $subview: true }
      ]
    };
  }
}
```

this in the button handler refers to the button itself, and **this.\$scope** references the Jet view^{[1](#)}.

Jet Links with Parameters

You can pass one or more parameters with a Jet link:

```
<!-- one -->
<a route="/details/data?id=2"></a>
<!-- or several -->
<a route="/details/data?id=2&name=some"></a>
```

2. Navigation with HTML links

Apart from navigating with links with the **route** attribute, you can create HTML links. This way is not so convenient, as the first one. You cannot prevent users from leaving the current view through an HTML link. In case there's no worries and you don't plan to guard users' unsaved data, you can create links with the **href** attribute. Suppose you have these view modules:

```
// views/demo.js
import {ToolbarView} from "toolbar";

const DemoView = {
  rows: [
    ToolbarView,
    { $subview: true }
  ]
};

// views/details.js
const DetailsView = {
  template: "App"
};
```

This is a link to the **Details** view as a subview of **DemoView**:

```
<!--index.html-->
<a href="#!/demo/details">Data</a>
```

You can pass one or more parameters in the link, e.g.:

```
<a href="#!/demo/details?id=2">Data</a>
<!-- or more -->
<a href="#!/demo/details?id=2&name=some">Data</a>
```

3. app.show()

Apart from links, you can use the **show** method of app to switch views. **app.show()** will rebuild the whole app or app module that called the method. A specific instance of the related view class is referenced with **this** if your handler is an *arrow function*². Reference app as **this.app** to call the **show** method from control handlers, for instance:

```
// views/toolbar.js
...
{ view:"button", value:"details", click: () => {
  this.app.show("/demo/details");
}}
```

After a button click, the URL will change, and the app UI will be rebuilt according to it.

app.show with URL parameters

You can pass one or more parameters with the URL:

```
// one
this.app.show("/demo/details?id=2");
// or many
this.app.show("/demo/details?id=2&name=some");
```

4. view.show()

You can also change the URL by calling the **show()** method from a specific view. A specific instance of the related view class is referenced with **this** from a handler that is defined as an *arrow function*³. Calling **show** from a view gives you more freedom, as it allows rebuilding only this view or only its subview, not the whole app or app module. For example, suppose you have a view like this:

```
// views/layout.js
...
cols:[
  { view:"button", value:"demo"},
  { $subview:true }
]
```

if the current URL is `/layout/details`, the subview is **details**. To replace **details** with a different subview, specify the name as it is or with `./`, and pass it to **this.show**:

```
// views/layout.js
...
{ view:"button", value:"demo", click: () => {
  this.show("demo");
}}

//or

{ view:"button", value:"demo", click: () => {
  this.show("./demo");
}}
```

The resulting URL is going to be `/layout/demo`.

If you want to rebuild the whole app and load the **demo** view as the only view, specify the name of the view with a slash:

```
// views/toolbar.js
...
{ view:"button", value:"demo", click: () => {
  this.show("/demo");
}}
...
```

Navigating to Upper Levels

The syntax of showing views resembles the way you navigate through directories. So you can move some levels up from `/layout/details`, for example:

```
{ view:"button", value:"bigview", click: () => {
```

```
        this.show("../bigview");  
    }}
```

As a result, the app URL will be */bigview*.

view.show with URL parameters

You can pass one or more parameters to show alongside the URL:

```
// one  
this.show("details?id=2");  
// or many  
this.show("details?id=2&name=some");
```

[1 ↑](#), [2 ↑](#), [3 ↑](#): To read more about how to reference apps and view classes, go to ["Referencing views"](#).

Popups and Windows

Working with Popups and Windows

Temporary views like popups and windows can be created with **this.ui**. **this.ui** takes care of the windows it creates and destroys them when their parent views are destroyed.

Consider a simple popup view:

```
// views/window.js
const win1 = {
  view: "popup",
  body: { template: "Text 1" }
};
export default win1;
```

Let's define a view class that will create this popup:

```
// views/top.js
import {JetView} from "webix-jet";

export default class TopView extends JetView {
  config(){
    return {
      cols:[
        { view:"form", width: 200, rows:[
          { view:"button", value:"Show Window 1" }
        ]},
        { $subview: true }
      ]
    };
  }
}
```

The popup is created in **init** of *TopView*:

```
// views/top.js
import win1 from "window";
...
init(){
  this.win1 = this.ui(win1);
}
```

this refers to *TopView*. **win1** becomes its child view. To show **win1**, call **this.win1.show()**. Here's the button click handler:

```
// views/top.js

{ view:"form", width: 200, rows:[
  { view:"button", value:"Show Window 1", click:(id) =>
    this.win1.show($$(id).$view) }
]
```

```
  ]}
```

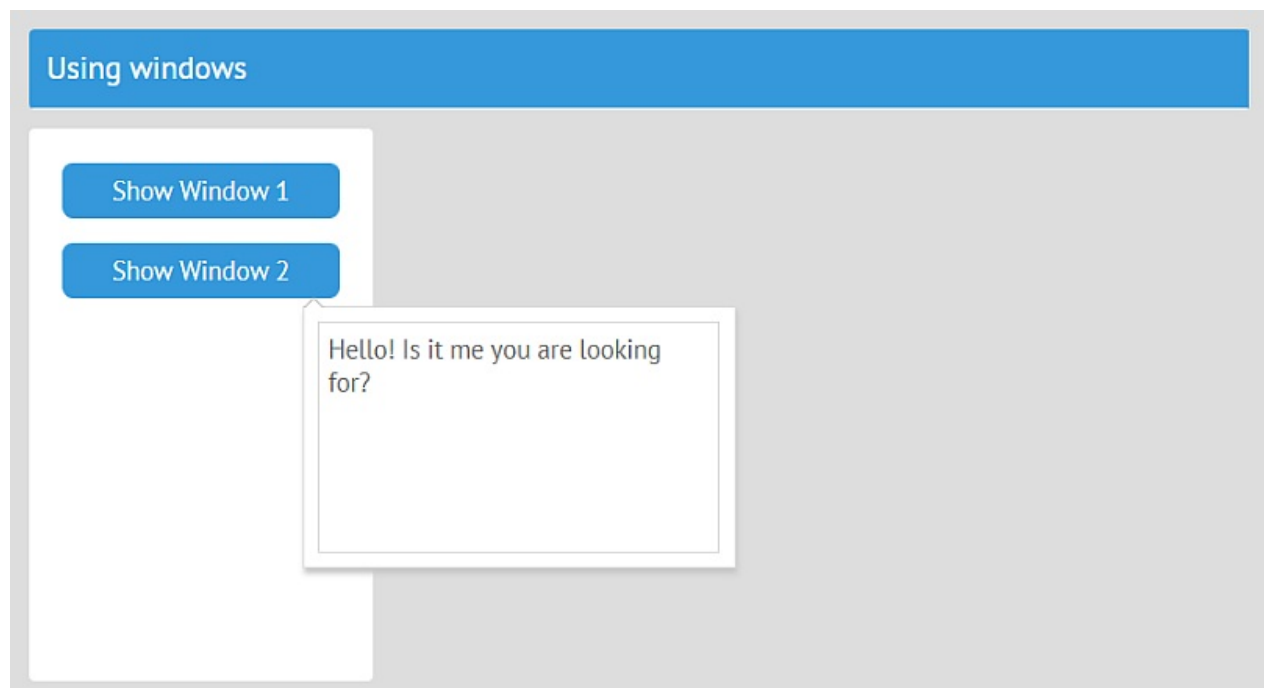
this.win1.show renders the popup at a position, defined by the parameter. In the code sample, the parameter is the HTML code of the button that calls the method, so the popup appears below the button. If you do not pass a parameter, **win1** will be rendered in the top left corner.

You can create windows and popups with view classes as well. Have a look at a similar popup, defined as a class:

```
// views/window2.js
import {JetView} from "webix-jet";

export class WindowsView extends JetView {
  config(){
    return {
      view:"popup",
      body:{ template:"Text 2" }
    };
  }
  show(target){
    this.getRoot().show(target);
  }
}
```

Unlike with a simple view, with a class view you have to redefine **show**. *this.getRoot* refers to the popup UI. So *this.getRoot().show(target)* does the same thing as *this.win1* from the previous example with a simple view **win1**. It shows the popup below the target view or component.



Here's how you initiate and show this popup (spoiler: exactly the same way as **win1**):

```
// views/top.js
import {JetView} from "webix-jet";
import WindowView from "window2";
```

```

export default class TopView extends JetView {
  config(){
    return {
      cols:[
        { view:"form", width: 200, rows:[
          { view:"button", value:"Show Window 2", click:(id) => {
            this.win2.show(id.$view) }
          ]},
        { $subview: true }
      ]
    };
  }

  init(){
    this.win2 = this.ui(WindowsView);
  }
}

```

[Check out the demo on GitHub >>](#)

Async views

Async (Promised) Views

Data is usually stored in a database on the server side. If the UI is built much faster than the data is loaded, it's a good idea to make use of promises to load the data. To make the UI wait for data, you can make an asynchronous request to a PHP script and give the UI a promise of data. Then the app will wait for data from a database, and only after a promise resolves, it will render the view with the data.

For example, there is a chart on the start page, and you need to define the colors of its lines, specified in the **series** parameter. Colors can be stored as inline data:

```
//views/statistics.js
import {JetView} from "webix-jet";

export class StatisticsView extends JetView {
  config() {
    return {
      view:"chart",
      series:[{ value:"#sales#", color:"#1293f8"},{value:"#sales2#", color:"#1293f8"}],
      url:"server/colors.php",
      ...//the rest of config
    }
  }
};
```

In practice, however, some UI configuration settings can be stored in a database. For example, you may want to store colors in a DB to allow the end-user to change them. In this case, a module can return a **promise** of the UI instead of the UI configuration.

A Promise Returned by config of a Class View

Let's use **webix.ajax** that makes an asynchronous request to a PHP script and returns a promise. After the promise resolves, the response is passed to a callback in **then**:

```
export class StatisticsView extends JetView {
  config() {
    return webix.ajax("server/colors.php").then(function(data){
      /* view creation */
      data = data.json();
      return {
        view:"chart",
        series:[
          { value:"#sales#", color:data[0].color},
          { value:"#sales2#", color:data[1].color}
        ]
      }
    });
  }
};
```

```
};
```

The **webix.ajax()** call sends an asynchronous request to the *server/colors.php* script on the server and returns a promise of data instead of real data. First, all the data should come to the client side and only after that the final view configuration will be constructed and the view will be rendered.

A Simple View as a Promise

The same view can be defined in a simpler way. You can define it as a promise of a view object:

```
export default webix.ajax("server/colors.php").then(function(data){
  /* view creation */
  data = data.json();
  return {
    view:"chart",
    series:[
      { value:"#sales#", color:data[0].color},
      { value:"#sales2#", color:data[1].color}
    ]
  };
});
```

Have a look at a list view defined as a promise:

```
const data = webix.ajax("data").then(res => {
  return {
    view:"list", options:res.json()
  }
});
export default data;
```

More explicitly, the same list view can be defined like this:

```
export default new Promise((res, rej) => {
  webix.ajax("data", function(text, res){
    const ui = {
      view:"list", options:res.json()
    };
    res(ui);
  })
});
```

Other Ways

There are two more ways to implement asynchronous data loading:

- **data.waitData** that is used for data components, such as DataCollection, List, Tree, DataTable, etc;
- **webix.promise** that allows treating the result of asynchronous operations without callbacks.

Referencing views

Referencing Views from Webix UI Events

In the new version of Webix Jet, there is a convenient reference to a Jet view from Webix UI events with the **this** pointer. Consider the following use-cases.

- [Reference to the View](#)
- [Reference to the App](#)
- [Reference to the Root UI Element of the View](#)
- [Referencing Parent Views and Subviews](#)
- [Referencing Webix Views and Controls](#)

1. Reference to the View

Due to ES6 *arrow functions*, you can refer to a Jet view with **this** from click handlers of Webix views. This way is shorter and advisable. For example, **this.show** in the handler of the button refers to **Toolbar**:

```
// views/toolbar.js

const Toolbar = {
  view: "toolbar",
  elements: [
    { view: "label", label: "Demo" },
    { view: "button", value: "details",
      click: () => {
        this.show("details");
      }
    }
  ]
};
export default Toolbar;
```

Another way to reference a Jet view is useful when you need to define a handler as *function*. In this case, **this** refers to a Webix view. Any Webix component created inside of a view has the **\$scope** property, which points to the Jet view. So if you want to change the URL from controls of the view, reference the view with **this.\$scope** and call its **show** method:

```
// views/toolbar.js

const Toolbar = {
  view: "toolbar",
  elements: [
    { view: "label", label: "Demo" },
    { view: "button", value: "details",
      click: function() {
        this.$scope.show(this.getValue());
      }
    }
  ]
};
```

```
export default Toolbar;
```

This is one of the cases when arrow functions do not help to shorten the syntax. Have a look at the same task done with an arrow function:

```
// views/toolbar.js

{ view: "button", value:"details",
  click: () => {
    this.show(this.getRoot().queryView({view:"button"}).getValue()
  }
}
```

`getRoot()` and `queryView()` are discussed [below](#).

2. Reference to the App

If you want to rebuild the app UI, you need to use `app.show("/new/url")`. You can shorten the syntax with an **arrow function** and reference the app with **this.app**:

```
// views/toolbar.js
import {JetView} from "webix-jet";

export default class ToolbarView extends JetView {
  config() {
    return {
      view: "toolbar",
      elements: [
        { view: "label", label: "Demo" },
        { view: "button", value:"details",
          click: () => {
            this.app.show("/demo/details");
            //or
            //this.app.show("/demo/" + this.getRoot().queryView({view:"button"}).getValue()
          }
        }
      ]
    };
  }
}
```

this.\$scope.app references the app that encloses the view if the handler is a *simple function*:

```
// views/toolbar.js
import {JetView} from "webix-jet";

export default class ToolbarView extends JetView {
  config() {
    return {
      view: "toolbar",
      elements: [
        { view: "label", label: "Demo" },
        { view: "button", value:"details",
          click: function () {
            this.$scope.app.show("/demo/"+this.getValue()
          }
        }
      ]
    };
  }
}
```



```

    }
  }
};
}

```

3. Reference to the Root UI Element of the View

Suppose you have a view with a form and you want to validate its input when users click **Submit**. To refer to the form itself and not the whole Jet view, use **this.getRoot()** inside an arrow function.

getRoot() references the root UI element returned by **config**, which is Webix Form in the code below:

```

// views/form.js
import {JetView} from "webix-jet";

export default class FormView extends JetView{
  config(){
    return {
      view:"form", elements:[
        { view:"text", name:"email", required:true, label:"E"},
        { view:"button", value:"save",
          click: () => {
            if (this.getRoot().validate())
              this.show("details");
          }
        }
      ]};
  }
}

```

After the form is validated, the form will be replaced with the **details** view.

If the handler is not an arrow function, refer to the form or any other root UI element as **this.\$scope.getRoot()**:

```

// views/form.js
{ view:"button", value:"save",
  click: function() {
    if (this.$scope.getRoot().validate())
      this.$scope.show("details");
  }
}

```

4. Referencing Parent Views and Subviews

JetView class has two methods to reference subviews (kids) and parent views.

1. getParentView()

getParentView() is used to reference the parent view of a subview. For example, there is a Jet view with a Webix list and a form as a static subview:

```

// views/listedit.js
import {JetView} from "webix-jet";
import form from "form";

```

```

export default class Parent extends JetView{
  config(){
    return {
      cols:[
        { view:"list", select:true },
        { $subview:form }
      ]
    }
  }
  getSelected(){
    this.getRoot.getSelectedItem();
  }
}

```

To get to the methods of Parent from the subview, you can use **this.getParentView()**:

```

// views/form.js
import {JetView} from "webix-jet";

export default class Child extends JetView{
  config(){
    return {
      view:"form", elements:[
        { view:"text", name:"name" }
      ]
    };
  }
  init(view){
    var item = this.getParentView().getSelected();
    view.setValues(item);
  }
}

```

When the parent and the subview are rendered, the form in the subview gets the name of the selected item in the list from the parent.

2. getSubview("name")

Use **getSubview()** to get to the methods of subviews from a parent. Consider an example with a parent and two static subviews:

```

// views/listedit.js
import {JetView} from "webix-jet";
import ChildList from "list";
import ChildForm from "form";

export default class ListEditView extends JetView{
  config(){
    return {
      cols:[
        { $subview:ChildList, name:"list" },
        { $subview:ChildForm, name:"form" }
      ]
    }
  }
}

```

```

    }
    ready(){
        var list = this.getSubview("list").getRoot();
        this.getSubview("form").bindWith(list);
    }
}

```

When all the views are ready (**ready** of the parent view is called after all its subviews are ready), the form is bound to the list. Mind that the parent view must have the **bindWith** method that calls the **bind** method of a Webix form:

```

// views/form.js
import {JetView} from "webix-jet";

export default class ChildForm extends JetView{
    config(){
        return {
            view:"form", elements:[
                { view:"text", name:"name" }
            ]
        };
    }
    bindWith(){
        this.getRoot().bind(widget);
    }
}

```

5. Referencing Webix Views and Controls

You already know how to change the URL by controls. Now have a look how the state of controls can be changed by the URL. For example, if you have a toolbar with a segmented button that is used to switch between two views:

```

// views/toolbar.js
import {JetView} from "webix-jet";

export default class ToolbarView extends JetView {
    config() {
        return {
            view: "toolbar",
            elements: [
                { view: "label", label: "Demo" },
                {
                    view: "segmented", options: ["details", "dash"],
                    click: function() {
                        this.$scope.show(this.getValue());
                    }
                }
            ]
        };
    }
}

```

This works fine, however, if you switch to **dash** and reload the page, the segmented button won't be

turned to **dash**. In more complex apps this behavior might be confusing¹. There is a way to solve this problem. You can reference a control inside a view and set its value in **init** of a class view that hosts the control.

1. queryView()

You can use **queryView** to reference the control. With this method, you can look for control by any attribute, like *view*, *name*, etc.

```
// views/toolbar.js
import {JetView} from "webix-jet";

export default class ToolbarView extends JetView {
  config() {
    /* same config with segmented button */
  }
  init(view, url) {
    if (url.length > 1)
      view.queryView({view:"segmented"}).setValue(url[1].page);
  }
}
```

If you switch to **dash** and reload the page now, the state of the button will be restored correctly.

queryView() is also often used for form validation. Consider also an example of simple form validation:

```
// views/form.js
import {JetView} from "webix-jet";

export default class FormView extends JetView{
  config(){
    return {
      view:"form", elements:[
        { view:"text", name:"email", label:"Email" },
        { view:"button", value:"save",
          click: () => {
            var text = this.getRoot().queryView({ name: '
              if (text.getValue())
                this.show("details");
            }
          }
        ]
      };
    }
  }
}
```

queryView will look for a view or a control with the *email* name inside the form. The same control can be found by its view type *text* or its label *Email*.

2. Getting by IDs

You can also use **this.\$\$("localId")** to reference a control and set its value in **init** of the parent Jet view. **this.\$\$("localId")** can be used to reference nested views and controls by their global or local

IDs.

```
// views/toolbar.js
import {JetView} from "webix-jet";

export default class ToolbarView extends JetView {
  config() {
    //...
    { view: "segmented", localId: "control", options: ["details"],
      click: function() {
        this.$scope.app.show("/demo/" + this.getValue());
      }
    }
  }
  init(ui, url) {
    if (url.length > 1)
      this.$$("control").setValue(url[1].page);
  }
}
```

Which way to reference controls and Webix views is better?

IDs must be unique, and the more developers are working on the project, the stronger odds are that someone will give the same IDs to another view.

One of the solutions is to give complex IDs, e.g. *"root:control"*:

```
// views/toolbar.js
...
{
  view: "toolbar",
  elements: [
    { view: "segmented", localId: "toolbar:segbtn", options: ["d",
      click: function() {
        this.$scope.app.show("/demo/" + this.getValue());
      }
    }
  ]
}
```

This will lessen the chances of non-unique IDs.

queryView is another solution to this problem. The syntax is longer, but it works as fine as the search by its local ID and lets you avoid IDs at all.

1: Actually, this task can be solved with the Menu plugin. You can [read about it in the dedicated section](#).

Models

Models

View modules return the UI configuration of the components. They describe the visual aspect of an application and shouldn't contain any data. For data, there is another type of modules - a **model**. Models are JS files that are placed in a separate folder. This division of UI and data is an advantage because if the data changes, all you have to do is change the data model. There's no need to modify view files.

There are several ways of loading data in Webix Jet:

- [shared models](#)
- [dynamic models](#)
- [remote models](#)
- [services for data](#)
- [models with webix.remote](#)

1. Shared Data

If you have some *relatively small data* and plan to use them in *many components*, you can create a model, initialize a data collection in it and load the data into the data collection. Here's an example of a data collection with static data:

```
// models/records.js
export const records = new webix.DataCollection({ data:[
  { id:1, title:"The Shawshank Redemption", year:1994, votes:678790},
  { id:2, title:"The Godfather", year:1972, votes:511495, rating:9.5},
  //...other records
]});
```

This is how the data is loaded from and saved to the server:

```
// models/records.js
export const records = new webix.DataCollection({
  url:"data.php",
  save:"data.php"
});
```

To use the data in a component, you need to parse it. You must parse data in **init**, not in **config** (leave *config* for UI only). Have a look at the example with a datatable:

```
// views/data.js
import {JetView} from "webix-jet";
import {records} from "../models/records";

export default class DataView extends JetView{
  config(){
    return { view:"datatable", autoConfig:true, editable:true };
  }
  init(view){
```

```

        view.parse(records);
    }
}

```

All the changes made in the datatable are saved to the server.

2. Dynamic Data

This is the model for *big data* (less than 10K records). These data can be *used only once* and mustn't be cached. The data can be loaded from a server with an AJAX request:

```

// models/records.js
export function getData(){
    return webix.ajax("data.php");
}

```

or from local storage:

```

// models/records.js
export function getData(){
    return webix.storage.local.get("data");
}

```

To parse data, pass *getData* to *view.parse*:

```

// views/data.js
import {JetView} from "webix-jet";
import {getData} from "../models/records";

export default class DataView extends JetView{
    config(){
        return { view:"datatable", autoConfig:true };
    }
    init(view){
        view.parse(getData());
    }
}

```

To save data, you can add one more function to the *records* model:

```

// models/records.js
export function getData(){
    return webix.ajax("data.php");
};
export function saveData(){
    return webix.ajax().post("data.php", { data });
}

```

In *init* you need to define a way to save data:

```

// views/data.js
import {JetView} from "webix-jet";
import {getData, saveData} from "../models/records";

export default class DataView extends JetView{
63

```

```

    config(){
        return { view:"datatable", autoConfig:true, editable:true };
    }
    init(view) {
        view.parse(getData());
        view.define("save", saveData);
    }
}

```

Loading and saving data can also be in **config** of the view module:

```

// views/data.js
import {JetView} from "webix-jet";
import {getData, saveData} from "../models/records";

export default class DataView extends JetView{
    config() {
        return {
            view:"datatable", autoConfig:true,
            url: getData,
            save: saveData
        }
    }
}

```

3. Remote Models for Huge Data

For really *huge data* (more than 10K records), you can use dynamic loading of Webix components and drop models :) Data will be loaded in portions when needed. For that, load data in the view code. You can do it with the **url** property. For saving data, use the **save** property.

```

// views/data.js
import {JetView} from "webix-jet";

export default class DataView extends JetView{
    config(){
        return {
            view:"datatable", autoConfig:true,
            url:"data.php",
            save:"data.php"
        };
    }
}

```

Data can also be loaded dynamically with the **load** method:

```

// views/data.js
import {JetView} from "webix-jet";

export default class DataView extends JetView{
    config(){
        var ui = { view:"datatable", autoConfig:true };
        ui.load("data.php");
        return ui;
    }
}

```



```
    }
}
```

Remote models are also convenient for prototyping.

Shared vs Dynamic vs Remote models

Let's recap main differences of the three ways to load and save data:

Model	Data Size	Usage
Shared	Small	Many times
Dynamic	Big	Once
Remote	Huge	Handy for prototyping

4. Services as Data Sources

You can use services instead of models as data sources. Suppose there's a list of customers and a grid that displays records on a selected customer. Here's how you can use a service that returns the ID of a selected list item:

```
var id = service.getSelected();
var data = records.getData(id);
```

A better and shorter way is:

```
var data = service.getNomenclature();
```

5. Using Webix Remote with Webix Jet

You can use [webix.remote](#) instead of sending AJAX requests. You must have a server-side script, e.g.:

```
<script src="api.php"></script>
```

The script can contain a *nm* class like this:

```
class Nomenc1 {
    public function getData(id) { /*...*/ }
}
$api->setClass("nm", new Nomenc1());
```

Here's a model that gets the class and all its methods:

```
// models/nomenc1.js
export default webix.remote.nm;
```

You can create a DataCollection:

```
// models/nomenc1.js
var data = new DataCollection({
    url: webix.remote.nm.getData
})
```

You can import the model and parse it into a view component:

```
// views/data.js
import records from "../models/nomencl";
...
init(view){
    view.parse(records.getData(id));
}
```

webix.remote is better than an AJAX request for several reasons.

First, you don't need to serialize data after loading. Compare the results of these two requests:

```
var data1 = webix.ajax("data/nomencl/154");           //{"id":154,"na
var data2 = webix.remote.nm.getNomenclature(154);      //{id:154,name:'
```

After receiving the response of the AJAX request, you have to call `JSON.parse(data1)` to turn a string into an object. The response of the second request is already an object.

Second, requests with *webix.remote* are safer due to CSRF-security.

And third, several requests are sent as one, which makes operations faster.

Further reading

For more details on services, read:

[-Services](#)

Routers

Routing

To manipulate the URL, views have Routers. Webix Jet has four predefined types of routers.

1. Hash Router (default)

The app URL is displayed after a hashbang. As this router is set by default, there's no need to define it in the config.

```
// myapp.js
var app = new JetApp({
  start: "/demo/details",
  router: JetApp.routers.HashRouter //optional
}).render();
```

2. URL Router

The app URL is displayed without a hashbang. Clandestine and cool, but there's a trick with this router. Your server-side code should be compatible.

Have a look at the example. Here are three views: one parent and two child views that are dynamically included by a click on jet links in the parent view:

```
// views/top.js
const TopView = {
  type: "space", rows: [
    { type: "header", template: "Url router",
    {
      type: "wide", cols: [
        { width: 200, css: "navblock", template: `
          <a route="/top/start"> - show start</a>
          <a route="/top/details"> - show details</a>
        ` },
        { $subview: true }
      ]
    }
  ]
};

const StartView = {
  template: "Start page"
};

const DetailsView = {
  template: "Details page"
};
```

Let's create an app from these views and choose `UrlRouter`:

```
// myapp.js
import {JetApp, UrlRouter} from "webix-jet";

webix.ready(() => {
  const app = new JetApp({
    id: "routers-url",
    router: UrlRouter,
    routerPrefix: "/routers-url", //!
    start: "/top/start"
  });
  app.render();
});
```

Note that there is a router prefix that is present in the URL instead of a hashbang. You must provide it if the app is hosted in a folder. In your *index.html* you should set the relative URL with the same prefix:

```
<!-- index.html -->
<script type="text/javascript">
  if(document.location.pathname == "/index.html")
    document.location.href = "/routers-url/";
</script>
```

Next, configure http redirects so that requests to all URLs triggered the html file of the app. This can be done through the webpack config:

```
// webpack.config.js
devServer:{
  historyApiFallback:{
    index : "routers-url.html"
  }
}
```

In the production app, it can be done through *apache/nginx* configuration.

[Check out the demo >>](#)

3. Store Router

With this guy, the app URL isn't displayed at all, but it is stored in the session storage. So no worries, you can still return to the previous and next views as if they are in the URL. This can be useful if you have a multilevel application (apps are subviews of other apps). The Store router is set for the enclosed apps because there's only one address bar and it's already taken by the outer app. Suppose you have closed an app module with a deep level of subviews and expect to be in the same place of this app when you switch to it again. The Store router allows this.

Here's an app module with a form view:

```
// app1.js
var app1 = new JetApp({
  start: "/form",
  router: JetApp.routers.StoreRouter
```

```
});
```

Next, the app module is included into a view and the view is included into another app:

```
// app2.js
const PageView = () => ({
  rows: [app1]
});

var app2 = new JetApp({
  start: "/page",
  router: JetApp.routers.HashRouter
}).render();
```

4. Empty Router

If you don't want to store the app part of the URL, there's the EmptyRouter for you. The app URL isn't displayed and isn't stored. It's used for nested apps because there is only one address bar.

```
// app1.js
var app1 = new JetApp({
  start: "/form",
  router: JetApp.routers.EmptyRouter
});
```

[Check out the demo >>](#)

5. Custom Routers

If these four routers aren't what you want, you can define your own.

Events and Methods

View Communication

Views are separated, but there should be some means of communication between them.

- [Parameters](#)
- [Events](#)
- [Methods](#)

Parameters

You can enable view communication with *parameters*. For instance, you need to open a form with some specific data from another view. Pass the needed parameters to *view.show*:

```
// views/data.js
export default class DataView extends JetView{
  config(){
    return { rows:[
      { $subview:true }
    ]};
  }
  init(){
    this.show("./form?id=1");
  }
}
```

And here is **form**:

```
// views/form.js
import {JetView} from "webix-jet";
import {getData} from "../models/records";

export default class FormView extends JetView{
  config(){
    return {
      view:"form", elements:[
        { view:"text", name:"name" },
        { view:"text", name:"email" }
      ]
    };
  }
  urlChange(view,url){
    if(url[0].params.id){
      this.getRoot().setValues( getData(id) );
    }
  }
}
```

In this simple example, as soon as DataView is initialized, the form is filled with data from a data

record with ID 1.

You can also pass several parameters to **show**:

```
this.show("./form?name=Jack&email=some");
```

Events

Feel free to use the in-app event bus for view communication.

Calling an Event

To trigger an event, call **app.callEvent**. You can call the method by referencing the app with **this.app** from an *arrow function*¹:

```
// views/data.js
import {JetView} from "webix-jet";

export default class DataView extends JetView{
  config(){
    return {
      view:"button", click:() => {
        this.app.callEvent("save:form");
      }
    }
  }
}
```

Attaching an Event

You can attach an event handler to the event bus in one view and trigger the event in another view.

This is how you can attach an event to a Jet view:

```
// views/form.js
import {JetView} from "webix-jet";

export default class FormView extends JetView{
  init(){
    this.app.attachEvent("save:form", function(){
      this.show("aftersave");
    });
  }
}
```

One more way to attach an event is **this.on** (**this** references a Jet view). This way is better, because the event is automatically detached when the view that called it is destroyed.

```
// views/form.js
import {JetView} from "webix-jet";

export default class FormView extends JetView{
  init(){
    this.on(this.app, "save:form", function(){
```

```

        this.show("aftersave");
    });
}
}

```

Once an event is attached, any other view can call it.

Declaring and Calling Methods

One more effective way of connecting views is methods. In one of the views, you can define a method, and in another view, we you can call this method.

Unlike events, methods can also return something useful. Methods can only be used for view communication when you know that a view with the necessary method exists. It's better to use this variant with a parent and a child views. A method is declared in the child view and is called in the parent one.

Events vs Methods

Have a look at the example. Here is a view that has the *setMode("mode")* method:

```

// views/child.js
import {JetView} from "webix-jet";

export default class ChildView extends JetView{
  config(){
    return {
      { view:"spreadsheet" }
    }
  }
  setMode("mode"){
    //sets the mode of the view
  }
}

```

And this is a parent view that will include *ChildView* and call its method:

```

// views/parent.js
import {JetView} from "webix-jet";
import ChildView from "child";

export default class ParentView extends JetView{
  config() {
    return {
      rows:[
        { view:"button", value:"Set mode", click:() => {
          this.getSubView().setMode("readonly")}
        },
        { $subview: ChildView }]
    }
  }
}

```

this.getSubView() refers to *ChildView* and calls the method. **getSubView()** can take a parameter with the name of a subview if there are several subviews, as you will see in the next section.

You can use methods for view communication in similar use-cases, but still events are more advisable here. Now let's have a look at the example where methods are better than events.

Methods vs Events

Suppose you want to create a file manager resembling Total Commander. The parent view will have two *file* views as subviews:

```
// views/manager.js
import FileView from "files";
...
config() {
  return {
    cols:[
      { name:"left", $subview:FileView },
      { name:"right", $subview:FileView }
    ]
  }
}
```

Here each subview has a name. *FileView* has the *loadFiles* method. Next, let's tell the file manager which paths to open in each file view:

```
// views/manager.js

init() {
  this.getSubView("left").loadFiles("a");
  this.getSubView("right").loadFiles("b");
}
```

Both subviews are referenced with **getSubView(name)**.

Further reading

For more info on view communications, you can read:

- [Services](#)

[2↑](#): To read more about how to reference apps and view classes, go to ["Referencing views"](#).

Services

Services

Services are one more means of view communication besides [the event bus and methods](#). Methods are preferable when you want to connect views with their children. This is impossible if views aren't related to each other, e.g. if you want to connect two subviews of the same view. This is where services help. You can create a service connected to one of the views, and other views can communicate with the view through the service. In short, a service here is created as shared functionality.

The same communication can be implemented with events, but this way is much more complex. With JS6, the problem can also be solved with creating and requiring a module. Services are better, because if you create two apps that require the same module, there's only one instance of the shared code. With services, a new instance of a service is created each time.

JetApp provides a means to initialize services. For example, you have a *masterTree* view and want other views to get the ID of a selected item. To set a service, call the **setService** method:

```
// views/tree.js
import {JetView} from "webix-jet";

export default class treeView extends JetView{
  config(){
    return { view:"tree" };
  }
  init() {
    this.app.setService("masterTree", {
      getSelected : () => this.getRoot().getSelectedId();
    })
  }
}
```

getSelected of the *masterTree* service returns the ID of the selected node of the tree. To call *getSelected*, use the **getService** method:

```
// views/form.js
import {JetView} from "webix-jet";
import {getData} from "../models/records";

export default class FormView extends JetView{
  config(){
    return {
      view:"form", elements:[
        { view:"text", name:"name" },
        { view:"text", name:"email" }
      ]
    };
  }
  init(){
```

```

        var id = this.app.getService("masterTree").getSelected();
        this.getRoot().setValues( getData(id) );
    }
}

```

Apart from view communication, services can be used for [loading and saving data](#).

Which Way of Views Communication To Choose?

	Used For	Direction	Receivers
Parameters	Initialization, Data loading		One
Events	Actions	Child->Parent	Many
Services	State Requests	Any	Many

Also remember that you can't use the same service for two instances of a view class, e.g. if you create a file manager with two identical file views. Use services if other ways aren't possible.

Tools

Tools

(to be continued)

Plugins

Plugins

The new version of Webix Jet provides both predefined plugins and the ability to create your own. This is the general syntax to use a plugin:

```
this.use(JetApp.plugins.PluginName, {
    id:"controlId", /* config
});
```

After the plugin name, you are to specify the configuration for the plugin, e.g. a control ID.

1. Default Plugins

- [the Menu plugin](#)
- [the UnloadGuard plugin](#)
- [the User plugin](#)
- [the Theme plugin](#)
- [the Locale plugin](#)
- [the Status plugin](#)

Menu Plugin

This plugin simplifies your life if you plan to create a menu. The plugin sets URLs for menu options, buttons or other controls you plan to use for navigation. Also, there is no need to provide handlers to restore the state of the menu on page reload or URL change. The right menu item is highlighted automatically.

My App!					
<div> <div>✉ DashBoard</div> <div>📁 Data</div> <div>News</div> </div>					
Title	Year	Votes	Rating	Rank	
The Shawshank Redempti	1994	678790	9.2	1	
The Godfather	1972	511495	9.2	2	
The Godfather: Part II	1974	319352	9	3	
The Good, the Bad and the	1966	213030	8.9	4	
My Fair Lady	1964	533848	8.9	5	
12 Angry Men	1957	164558	8.9	6	

Let's create a toolbar with a segmented button and use the plugin for it:

```
// views/toolbar.js
import {JetView} from "webix-jet";

export default class ToolbarView extends JetView{
  config(){
    return {
      view:"toolbar", elements:[
        { view:"label", label:"Demo" },
        { view:"segmented", localId:"control", options:[
          "details",
          "dash"
        ]}
      ]};
  }
  init(ui, url){
    this.use(JetApp.plugins.Menu, {
      id:"control"
    });
  }
}
```

The plugin config contains the ID of the view element that is going to serve as a menu. If you click the buttons and reload the page, the app will behave as expected. The **Menu** plugin has one more good part. You can change the URLs for every menu item. Let's set URLs for the buttons in the plugin config:

```
// views/toolbar.js
import {JetView} from "webix-jet";

export default class ToolbarView extends JetView {
  config(){
    return {
      view:"toolbar", elements:[
        { view:"label", label:"Demo" },
        { view:"segmented", localId:"control", options:[
          "details",
          "dash"
        ]}
      ]};
  }
  init(ui, url){
    this.use(JetApp.plugins.Menu, {
      id:"control",
      urls:{
        details : "demo/details",
        dash : "demo/dash"
      }
    });
  }
}
```

[Check out the demo >>](#)

UnloadGuard Plugin

The **UnloadGuard** plugin can be used to prevent users from leaving the view on some conditions. For example, this can be useful in the case of forms with unsaved or invalid data. The plugin can intercept the event of leaving the current view and, e.g. show the *are you sure* dialogue.

The screenshot shows a web form with the following fields: 'Event name *' (text input), 'Start Date *' (date/time picker showing '14:25 Sat, 04 Nov'), 'End Date *' (date/time picker showing '15:25 Sat, 04 Nov'), 'All-day' (checkbox), 'Address *' (text input), 'Calendar *' (dropdown menu showing 'My Cal'), and 'Details' (text area). A modal dialog box titled 'Are you sure ?' is centered over the form, with 'OK' and 'Cancel' buttons. At the bottom of the form are 'Reset' and 'Save' buttons.

The plugin reacts to an attempt of changing the URL. The syntax for using a plugin is `this.use(plugin,handler)`. Use takes two parameters:

- a plugin name
- a function that will handle the **Unload** event

You can move validation from the **Save** button handler to the plugin handler. Let's have a look at a form with one input field:

```
import {JetView} from "webix-jet";
export default class FormView extends JetView {
  config(){
    return {
      view:"form", elements:[
        { view:"text", name:"email", required:true, label:"Email" },
        { view:"button", value:"save", click:() => this.showSaveDialog() }
      ]
    };
  }
  init(ui, url){
    this.use(JetApp.plugins.UnloadGuard, () => {
      if (this.getRoot().validate())
        this.showSaveDialog();
    });
  }
}
```

```

        return true;

    return new Promise((res, rej) => {
        webix.confirm({
            text: "Are you sure ?",
            callback: a => a ? res() : rej()
        })
    });
});
}
}

```

If the input isn't valid, the function returns a promise with a dialogue window. Depending on the answer, the promise either resolves and the Guard lets the user go to the next view, or it rejects. No pasaran.

[Check out the demo >>](#)

User Plugin

The *User* plugin is useful if you create apps that need authorization. When the plugin is included, the **user** service is launched. Let's look how to use the plugin with a custom script.

Login through a custom script

The plugin uses a **session** model, [check it out](#). It contains requests to *php* scripts for logging in, getting the current status, and logging out. In the code, there are following functions:

- **status** - returns the status of the current user:

```

// models/session.js
function status(){
    return webix.ajax().post("server/login.php?status")
        .then(a => a.json());
}

```

- **login** - logs the user in, returns an object with his/her access rights settings, a promise of this object or *null* if something went wrong. The parameters are:
- *user* - username;
- *pass* - password.

```

// models/session.js
function login(user, pass){
    return webix.ajax().post("server/login.php", {
        user, pass
    }).then(a => a.json());
}

```

- **logout** - logs the user out:

```

// models/session.js
function logout(){

```



```

        return webix.ajax().post("server/login.php?logout")
            .then(a => a.json());
    }

```

To use a plugin, call `app.use` and pass the **session** model to it:

```

// myapp.js
import session from "models/session";
...
app.use(plugins.User, { model: session });

```

Here's an example of a form for logging in, which can be included into a view class:

```

// views/login.js
const login_form = {
    view: "form",
    rows: [
        { view: "text", name: "login", label: "User Name", labelPosition: "before" },
        { view: "text", type: "password", name: "pass", label: "Password", labelPosition: "before" },
        { view: "button", value: "Login", click: function() {
            this.$scope.do_login();
        }, hotkey: "enter" }
    ],
    rules: {
        login: webix.rules.isNotEmpty,
        pass: webix.rules.isNotEmpty
    }
};

```

The form has two validation rules.

To implement the way to log in, you can use the **login** method of the *User* plugin.

```
login(name, password)
```

login receives the username and the password, verifies them and if everything's fine, shows the *afterLogin* page. Otherwise, it shows an error message.

Here's how the **do_login** method is implemented:

```

// views/login.js
import {JetView} from "webix-jet";

export default class LoginView extends JetView{
    ...
    do_login(){
        const user = this.app.getService("user");
        const form = this.getRoot().queryView({ view: "form" });

        if (form.validate()){
            const data = form.getValues();
            user.login(data.login, data.pass).catch(function(){
                //error handler
            });
        }
    }
}

```

```

    }
  }
  ...
}

```

If users submit their name and password, *user.login* is called. You can add an error handler for an invalid username and password.

Related demo:

- The [complete login.js file](#);
- The [demo on logging in with custom scripts](#).

The **User** plugin has other useful methods.

`getUser()`

getUser returns the data of the currently logged in user.

`getStatus()`

getStatus returns the current status of the user. It can receive an optional boolean parameter *server*. The **User** service checks every 5 minutes the current user status and warns a user if the status has been changed. For example, if a user logged in and didn't perform any actions on the page during some time, the service will check the status and warn the user if it has been changed.

`logout()`

logout ends the current session and shows an afterLogout page, usually it's the login form.

Login with an external OAuth service (Google, Github, etc.)

Theme plugin

This is a plugin to change app themes. The plugin launches the **theme** service. There are two methods that the service provides:

`getTheme()`

The method returns the name of the current theme.

`setTheme(name)`

The method takes one obligatory parameter - the name of the theme - and sets the theme for the app.

Consider an example. The service locates links to stylesheets by the *title* attribute. Here are the stylesheets for the app:

```

// index.html
<link rel="stylesheet" title="flat" type="text/css" href="//cdn.webio...
<link rel="stylesheet" title="compact" type="text/css" href="//cdn.w

```

Next, you need to provide a way for users to choose themes. Here's a view with a segmented button:

```

// views/settings.js

```

```
import {JetView} from "webix-jet";

export default class SettingsView extends JetView {
  config(){
    return {
      type:"space", rows:[
        { template:"Settings", type:"header" },
        { name:"skin", optionWidth: 120, view:"segmented", label:"Theme",
          {id:"flat-default", value:"Default"},
          {id:"flat-shady", value:"Shady"},
          {id:"compact-default", value:"Compact"}
        ], click:() => this.toggleTheme() /* not implemented
      {}
    ]
  };
}
}
```

Note that option IDs should have two parts, the first of them must be the same as the *title* attribute of the link to a stylesheet. The **theme** service must get the theme name, chosen by a user. Then it locates the correct stylesheet and sets the theme. Let's add a handler for the segmented button and define it as a class method:

```
// views/settings.js
import {JetView} from "webix-jet";

export default class SettingsView extends JetView {
  ...
  toggleTheme(){
    const themes = this.app.getService("theme");
    const value = this.getRoot().queryView({ name:"skin" }).currentValue;
    themes.setTheme(value);
  }
}
```

Inside the method, the **theme** service is launched. Then **queryView** located the segmented by its name and gets the user a choice. After that, the service sets the chosen theme by adding a corresponding CSS class name to the body of the HTML page.

To restore the state of the segmented button after the new theme is applied, you need to get the current theme in the class **config** and add the **value** property of the segmented setting it to the correct value:

```
// views/settings.js
const theme = this.app.getService("theme").getTheme();
...
{ name:"skin", optionWidth: 120, view:"segmented", label:"Theme",
  {id:"flat-default", value:"Default"},
  {id:"flat-shady", value:"Shady"},
  {id:"compact-default", value:"Compact"}
], click:() => this.toggleTheme(), value:theme }
...
}
```

[Check out the complete code >>](#)

Locale plugin

This is a plugin for localizing apps. Locale files are usually created in the *locales* folder. This is an example of the Spanish locale file:

```
// locales/es.js
export default {
  "Settings" : "Ajustes",
  "Language" : "Idioma",
  "Theme" : "Tema"
};
```

The Locale plugin launches a service. The **locale** service has several methods for localizing apps.

the `_(value)` helper

This helper looks for the field in a locale file and returns the translated value. E.g. `this.app.getService("locale")._("Settings")` will return *"Ajustes"*, if Spanish is chosen. If you need to localize a lot of text, it's reasonable to create a shorthand for the method:

```
const _ = this.app.getService("locale")._;
```

`getLang()`

The method returns the current language.

`setLang(name)`

The method sets the language passed by the name of the locale file. It also calls **app.refresh** and re-renders all the views.

Consider an example:

```
import {JetApp, JetView, plugins} from "webix-jet";

export default class SettingsView extends JetView {
  config(){
    const _ = this.app.getService("locale")._;
    const lang = this.app.getService("locale").getLang();

    return {
      type:"space", rows:[
        { template:_("Settings"), type:"header" },
        { name:"lang", optionWidth: 120, view:"segmented", list:
          [{id:"en", value:"English"},
           {id:"es", value:"Spanish"}
          ], click:() => this.toggleLanguage(), value:lang },
        {}
      ]
    };
  }
  toggleLanguage(){
    const langs = this.app.getService("locale");
    const value = this.getRoot().queryView({ name:"lang" }).getVa
```

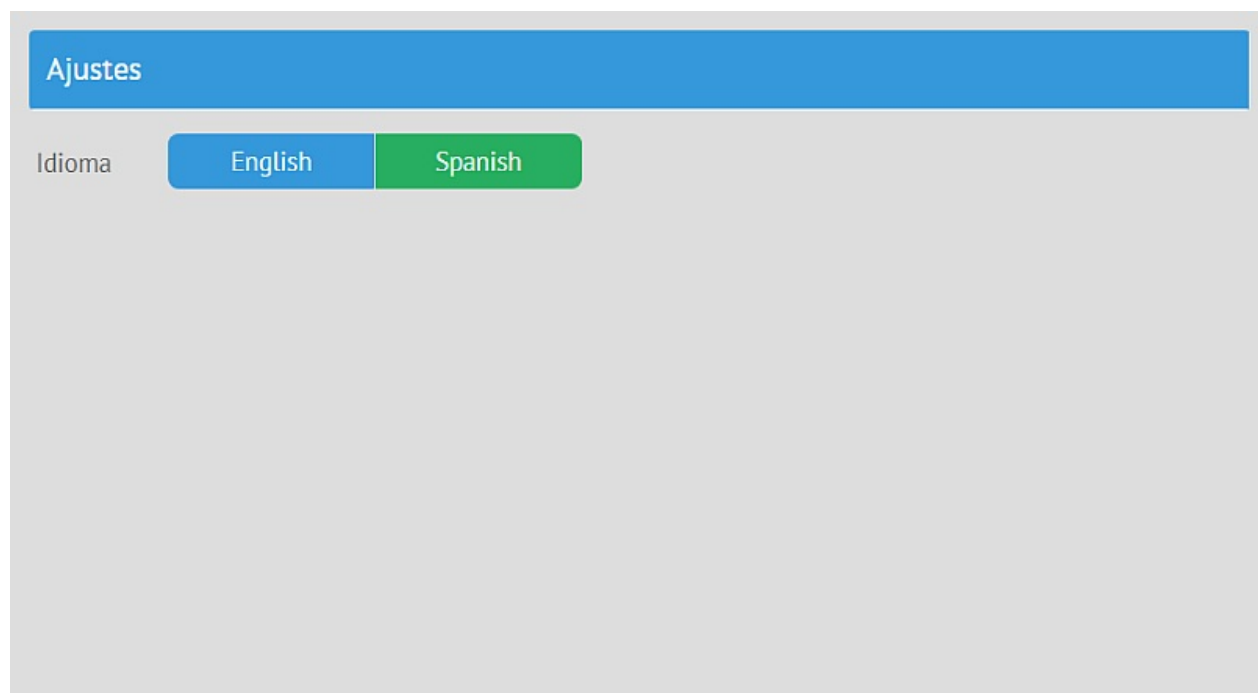
```

        langs.setLang(value);
    }
}

```

When a user chooses a language, a corresponding file is located and the app language is changed. IDs of the language options should be the same as the locale file names (e.g. "es", "en"). **getLang** in config returns the current locale, which is used to restore the value of the segmented button.

[Check out the demo >>](#)



Status Plugin

This plugin is useful if you want to show the status of data loading in case it takes time, to confirm success or to show an error message. These are the status messages that you can see:

- "Ok"
- "Error"
- "Connecting..."

This is how you can include the plugin:

```

// views/data.js
...
config(){
    return {
        rows:[
            { id:"app:status", view:"label" }
        ]
    };
}
init(){
    this.use(plugins.Status, {
        target: "app:status",
        ajax:true,

```

```
        expire: 5000
    });
}
```

The **target** property is the ID of the view component where you want to display the message (a label in this example). *ajax:true* enables asynchronous requests. *expire* defines the time after which the status message disappears (5 seconds in this case). By default, the time is set to 3 seconds, and if you set it to 0, the status message will stay as long as the page is open.

Status can have two more properties in its config:

- **data** - a property that defines the ID of the data component to track.
- **remote** - a boolean property that enables *webix.remote* - a protocol that allows the client component to call functions on the server directly.

[Check out the demo >>](#)

Some Data				
Title	Year	Votes	Rating	Rank
The Shawshank Redemption	1994	678790	9.2	1
The Godfather	1972	511495	9.2	2

2. Custom Plugins

You can define your own plugins.

Inner Events and Error Handling

Inner Jet Events and Error Handling

- [Inner Events](#)
- [Error Handling](#)

Apart from using built-in means like plugins, you can use a number of inner events. You might need this for adding some functionality if the plugins aren't enough or for error handling and debugging.

Inner Events

- [app:render](#)
- [app:route](#)
- [app:guard](#)

app:render

The event is triggered before each view of an app is rendered. You can use it to change the UI config that you defined and add properties to UI controls. For instance, here is how you can disable buttons:

```
// myapp.js

app.attachEvent("app:render", function(view,url,result){
    if (result.ui.view === "button")
        result.ui.disabled = true;
});
```

The event receives three parameters:

- **view** - the view for which the event is called (this.\$scope)
- **url** - the URL as an array of URL elements
- **result** - a wrapper object for UI; it's created in case you want to change the UI (e.g. to put it into some other view)

For example, you can put all the buttons on a toolbar:

```
// myapp.js

app.attachEvent("app:render", function(view,url,result){
    if (result.ui.view === "button")
        result.ui = {
            view:"toolbar", rows:[ result.ui ]
        };
});
```

app:route

Handling the **app:route** event resembles redefining the **urlChange** of a class view. The event fires after navigation to a view and can be used to send notifications or sending messages to a log:

```
// myapp.js

app.attachEvent("app:route", function(url){
    webix.ajax("log.php", url);
})
```

app:route receives one parameter - the URL as an array of URL elements.

app:route is used by the *Menu* plugin to highlight menu options according to the URL.

app:guard

The **app:guard** event is triggered before navigation to another view. One of the typical cases to use this event is to create a guard: block some views and redirect users somewhere else. The **app:guard** event is called by the *UnloadGuard* plugin. You can attach **app:guard** with:

```
// myapp.js

app.attachEvent("app:guard", function(url, view, nav){
    if (url.indexOf("/blocked") !== -1){
        nav.redirect = "/top/allowed";
    }
})
```

The event handler receives three parameters:

- *url* - a string with the attempted URL
- *view* - the parent view that contains a subview from the URL
- *nav* - an object that defines navigation to the next view

nav has three properties:

- *redirect* is the new URL; in the example above, it's used for redirection
- *url* is the URL split into an array of views
- *confirm* is a promise that is resolved when the **app:guard** event is called

Suppose you have a layout with three views, one parent and two subviews (simple template views for the example). This is the parent view that has two buttons that call **show** to render subviews:

```
// views/top.js
import {JetView} from "webix-jet";
import allowed1 from "allowed";
import blocked from "blocked";

export default class TopView extends JetView {
    config(){
        return {
            rows:[
                { type:"wide",cols:[
                    { view:"form", width: 200, rows:[
                        { view:"button", value:"Allowed page", c:
                            this.show("allowed") },
                        { view:"button", value:"Blocked page", c:
                            this.show("blocked") }
                    ]
                }
            ]
        }
    }
}
```



```

    }},
    { $subview: true }
  ]
}]]
};}}

```

One of the subviews is supposed to be blocked. Let's create a guard that will block it and redirect users to the *allowed* subview. Group the views into app and attach the **app:guard** event:

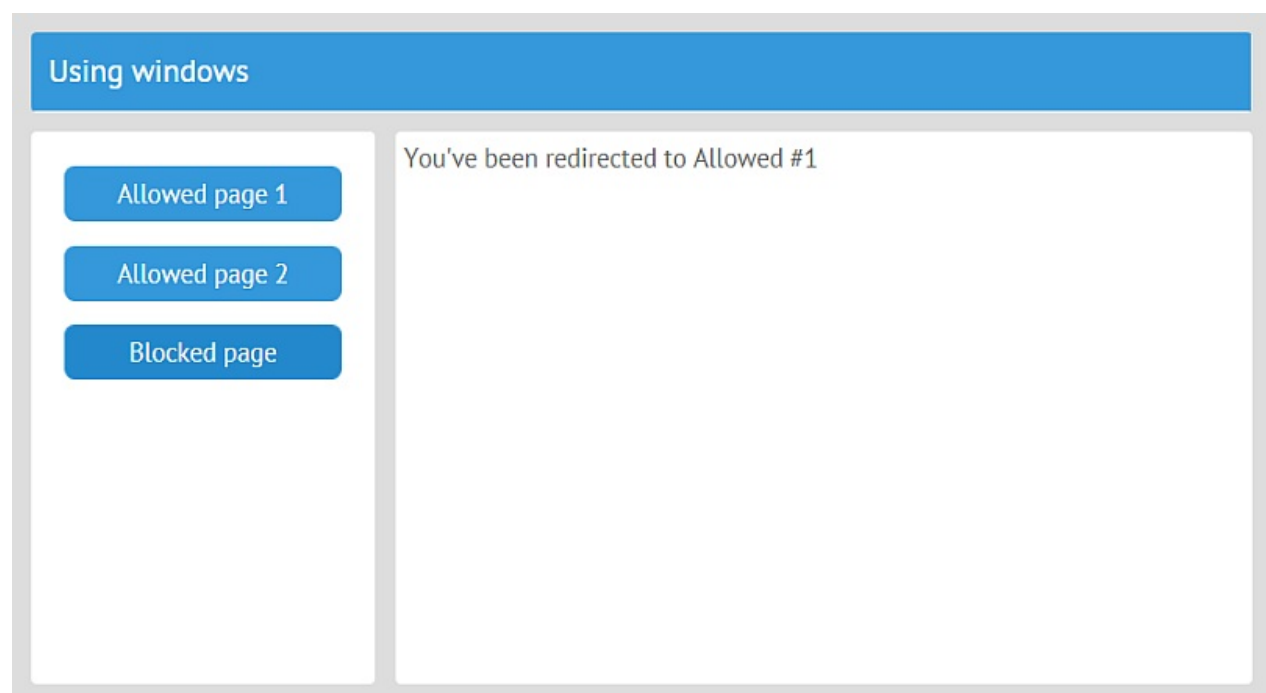
```

// myapp.js
import {JetApp} from "webix-jet";

const app = new JetApp({
  start: "/top/blocked"
});
app.attachEvent("app:guard", function(url, view, nav){
  if (url.indexOf("/blocked") !== -1){
    nav.redirect = "/top/allowed";
  }
});
app.render();

```

[The demo is available on Github >>](#)



Error Handling and Debugging

There are four events that can be used to handle errors.

- [app:error](#)
- [app:error:resolve](#)
- [app:error:render](#)
- [app:error:initview](#)

app:error

This is a common event for all errors. The errors are logged if you set the **debug** property in your app config:

```
// myapp.js
import {JetApp} from "webix-jet";

var app = new JetApp({
  debug: true // console.log and debugger on error
});
```

Besides logging errors, this will enable a debugger.

You can also do something else, for example, show an error message in an alert box:

```
// myapp.js

app.attachEvent("app:error", function(err){
  alert("Error");
});
```

The event receives one parameter - the error object.

Younger Siblings of *app:error*

Besides the common error event, there are three events for specific error types.

Useful for End-Users

app:error:resolve fires when Jet can't find a module by its name. If this happens, it would be useful to redirect users somewhere else instead of showing them an empty screen:

```
// myapp.js

app.attachEvent("app:error:resolve", function(err, url) {
  webix.delay(() => app.show("/some"));
});
```

app:error:resolve receives two parameters:

- **err** - an error object
- **url** - the URL

Useful for Developers

These error events are more useful for developers as they inform about errors related to the UI.

app:error:render

This event is triggered on errors during view rendering, mostly Webix UI related. It means that some view UI config has been written incorrectly.

```
// myapp.js
```

```
app.attachEvent("app:error:render", function(err){
    alert("Check UI config");
});
```

The event receives one parameter - the error object.

```
app:error:initview
```

This event is triggered in the case of an error during view rendering, mostly Webix Jet related. It means that Jet, while rendering Webix UIs, was unable to render the app UI correctly.

```
// myapp.js
```

```
app.attachEvent("app:error:initview", function(err,view){
    alert("Unable to render");
})
```

The event takes two parameters:

- **err** - the error object
- **view** - the view that caused the error

Jet Recipes

Jet Recipes

In this section you'll find elegant solutions for some common tasks.

- [Access control on view level](#)
- [Access control on app level](#)
- [Responsive UI](#)

Access Guard - View Level

Task: You want to create views with several levels of access for different groups of users.

Solution: Just add a check in config.

For example, you have two user groups: *readers* and *writers*. You want to display some of the app contents depending on the group. This is the UI component with contents that you want to show to *readers*:

```
export default class limited extends JetView{
    config(){
        return {
            view:"form", rows:[
                { label:"Name", view:"text" },
                { label:"Email", view:"text" },
                {}
            ]};
        }
    }
}
```

And this is the version of the same component you want to show to *writers*:

```
export default class limited extends JetView{
    config(){
        return {
            view:"form", rows:[
                { label:"Name", view:"text" },
                { label:"Email", view:"text" },
                { view:"text", label:"Salary" },
                { view:"button", value:"Delete" },
                {}
            ]};
        }
    }
}
```

The solution is to create one common view and to *configure* its UI depending on the user group. The user group of the current group will be stored in the app config. Let's add a property with a default group name into the app config:

```
// myapp.js
import limited from "views/limited";

const app = new JetApp({
  access:      "reader",
  start:       "/top/limited"
});
```

top is a view with a side menu and a toolbar that includes *limited* as its subview. Next, let's define the UI for **limited** for default users (*readers*):

```
// views/limited.js
import {JetView} from "webix-jet";

export default class limited extends JetView{
  config(){
    var ui = { view:"form", rows:[
      { label:"Name", view:"text" },
      { label:"Email", view:"text" },
      {}
    ]};

    return ui;
  }
}
```

Next, let's add a check in config and change the view config for *writers*:

```
// views/limited.js
import {JetView} from "webix-jet";

export default class limited extends JetView{
  config(){
    var ui = { view:"form", rows:[
      { label:"Name", view:"text" },
      { label:"Email", view:"text" }
    ]};

    if (this.app.config.access == "writer"){
      ui.rows.push({ view:"text", label:"Salary" });
      ui.rows.push({ view:"button", value:"Delete" });
    }

    ui.rows.push({});

    return ui;
  }
}
```

this.app.config.access gets to the access level of the current user. If you use [the User plugin](#), you can get the user name with *getUser* and check the group. For the demo, the user group is changed by a control in *top*:

```
// views/top.js
```

```
import {JetView} from "webix-jet";

export class TopView extends JetView {
  config(){
    return {
      type:"space", rows:[
        { view:"toolbar", cols:[
          { view:"segmented", label:"Access Level",
            value:this.app.config.access,
            options:["reader","writer"], click:function()
              // change access level, for demo only
              var app = this.$scope.app;
              app.config.access = this.getValue();
              webix.delay(function(){
                app.refresh();
              });
          }
        ]},
        { $subview: true }
      ]};
  }
}
```

`webix.delay` calls **refresh** in 1 ms to ensure that the config has been reset before repainting the app.

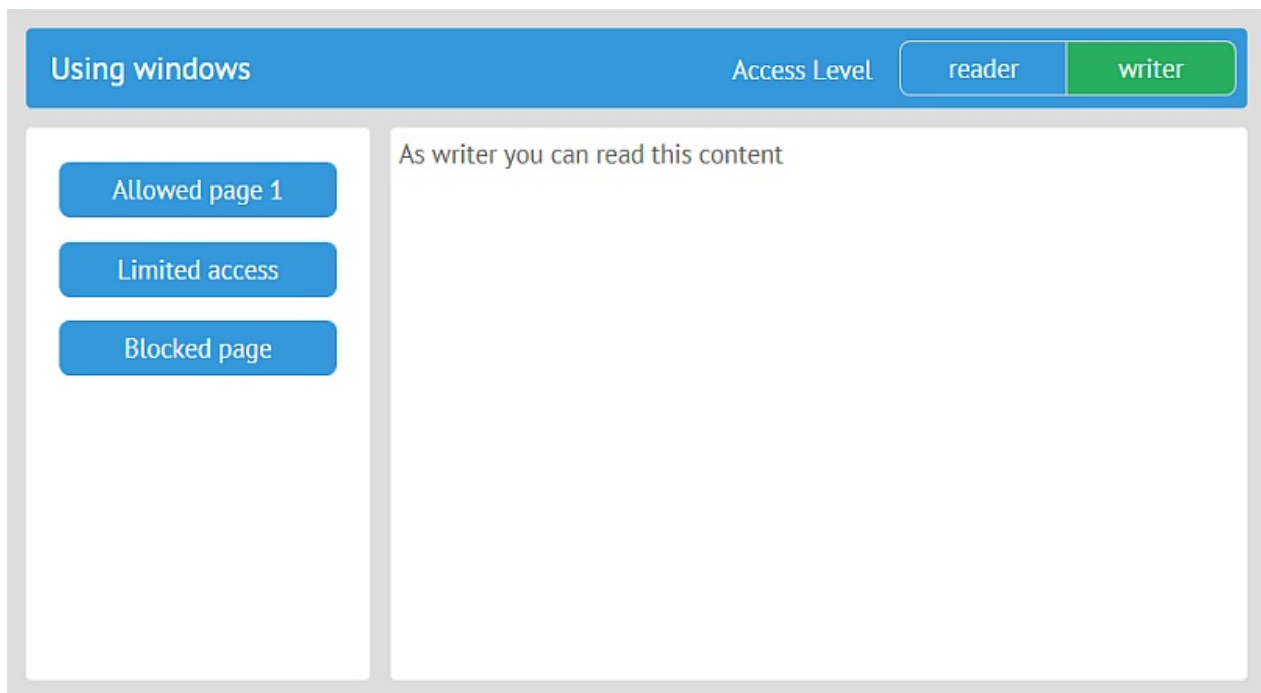
Here's an example how to block a view for *readers*:

```
// views/blocked.js
import {JetView} from "webix-jet";

export default class blocked extends JetView{
  config(){
    if (this.app.config.access !== "writer"){
      return { };
    }

    return { template:"As writer you can read this content" };
  }
}
```

You can show this view via `top/blocked`.



[Check out this solution on GitHub >>](#)

Access Guard -- App Level

Task: You want to create views with several levels of access for different groups of users.

Solution: use **app:guard** event.

The **app:guard** event is triggered before navigation to another view. Here's how you can attach **app:guard**:

```
// myapp.js
...
app.attachEvent("app:guard", function(url, view, nav){
    if (url.indexOf("/blocked") !== -1)
        nav.redirect="/somewhere/else";
})
```

The event handler receives three parameters:

- *url* - a string with the attempted URL
- *view* - the parent view that contains a subview from the URL
- *nav* - an object that defines navigation to the next view

nav has three properties:

- *redirect* is the new URL; in the example above it's corrected by the guard
- *url* is the URL split into an array of URL elements
- *confirm* is a promise that is resolved when the **app:guard** event is called

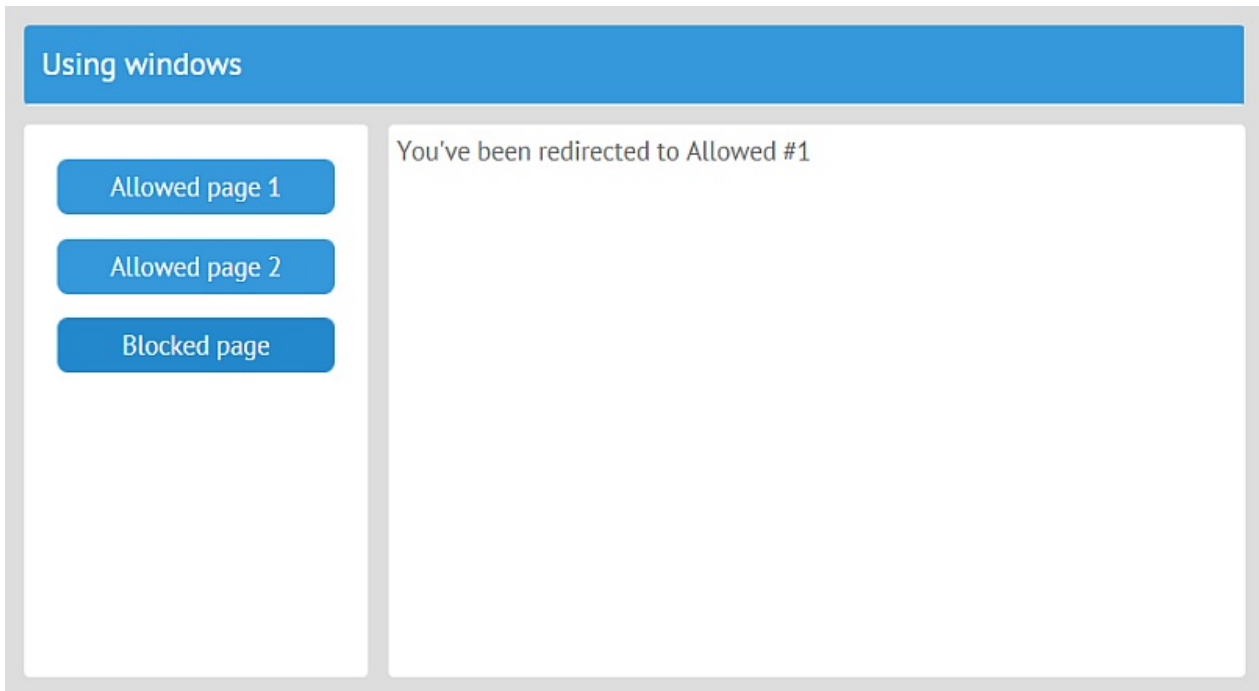
Here's how you can block a view and redirect users to some *allowed* subview:

```
// myapp.js
const app = new JetApp({
```

```

    start:          "/top/blocked"
  });
  app.attachEvent("app:guard", function(url, view, nav){
    if (url.indexOf("/blocked") !== -1){
      nav.redirect = "/top/allowed";
    }
  });
  app.render();

```



[Check out the solution on Github >>](#)

Responsive UI - Sizing UI to Device

Task: You want to create a responsive UI.

Solution:

1. ...is partly available out of the box (by Webix UI)

Webix UI resizes components automatically if you don't set fixed sizes. If you minimise your browser window or open the app from a smartphone, components will shrink. However, this is not enough if you want to reorganize your components, e.g. if for wide screens you want to display the layout in columns and for narrower screens in rows or in tabs or simply display less content.

1. Just add a check in config :)

Okay, suppose you want to distinguish two types of screens: *small* (less than 800px) and *wide*. (800 is just a number, you can choose the one you suppose is right). And there are two datatables (ListA and ListB) you want to display in two columns for *wide* screens and in two tabs for *small* screens. *StartView* is the layout. Add a property to app config and initialize it with a function that will count the width of the screen:

```

// myapp.js
webix.ready(() => {

```



```

    const app = new JetApp({
      start:      "/start"
    });
    const size = () => document.body.offsetWidth > 800 ? "wide" : "small";
    app.config.size = size();
    app.render();
  });

```

This ensures that when the app is rendered, the right size is calculated. This, nevertheless, doesn't solve dynamic resizing. Yet. This is what will make the app UI responsive and dynamic:

```

// myapp.js
webix.ready(() => {
  const app = new JetApp({
    //config
  });

  const size = () => document.body.offsetWidth > 800 ? "wide" : "small";
  app.config.size = size();
  webix.event(window, "resize", function(){
    var newSize = size();
    if (newSize !== app.config.size){
      app.config.size = newSize;
      app.refresh();
    }
  });

  app.render();
});

```

webix.event attaches an event handler to a browser window. On window resize, the screen type will be recalculated and changed if necessary. Don't forget to **refresh** the app.

Responsive Layout

Here's how **size** defines the layout (**StartView**). For small screens, the grids will be put in tabs, and for wide screens they will be put side by side:

```

// views/start.js
import {JetView} from "webix-jet";

export default class StartView extends JetView {
  config(){
    switch(this.app.config.size){
      case "small":
        return {
          view:"tabview", tabbar:{ optionWidth:100 }, cells:
            [
              { body: { rows:[ ListA ]}, header:"Table 1" },
              { body: { rows:[ ListB ]}, header:"Table 2" }
            ]
        };
      case "wide":
        return {

```

```

        type:"space", cols:[
            ListA,
            ListB
        ]
    };
}
}
}

```

Table 1		Table 2			
Title	Year	Votes	Rating	Rank	
The Shawshank Redemption	1994	678790	9.2	1	
The Godfather	1972	511495	9.2	2	

Responsive Content

One more way to make your app responsive is to display smaller content for small screens. Let's leave only two columns in one of the datatables for small screens:

```

// views/listb.js
import {JetView} from "webix-jet";

export default class ListB extends JetView {
    config(){
        var config = {
            view:"datatable",
            editable:true
        };

        switch(this.app.config.size){
            case "small":
                config.columns = [
                    { id:"id" },
                    { id:"title", fillspace:true }
                ];
                break;
        }
    }
}

```

```
        default:
            config.autoConfig = true;
            break;
    }

    return config;
}
init(view){
    view.parse(data); //a data collection
}
}
```

Table 1		Table 2
id	title	
1	The Shawshank Redemption	
2	The Godfather	

[Check out the solution on GitHub >>](#)