Contents

Chapter 1

Clove Manual

1.1 License

clove is written by Josef Hahn under the terms of the GPLv3 or higher.

Please read the `LICENSE` file from the package and the Dependencies section for included third-party stuff.

1.2 About

Clove is a user interface library for the web, which offers a powerful browser-side JavaScript toolkit for composing rich and neat web application frontends.

Clove is designed to get work done.

- **Integrating Clove is easy!** Just one or two `script`s and one `style` to be added to your existing html before you can begin.
- **The programming interface is easy!** No esoteric tricks for simple things. It's 100% standard JavaScript, running inside the browser, with a clean api.
- **A rich and complete set of widgets is included**. This goes from labels and buttons to data views like tables and trees. It's easy to implement custom widgets.
- All widgets can be used **either stand-alone in your existing document flow or completely managed** by the powerful Clove layouting system.
- **It works in all modern web browsers**, mobile and desktop, and can help realizing applications working great in both worlds.
- Beyond widgets, Clove comes with many kind of versatile tools for typical tasks, like internationalization, branding, ...
- It's well documented. Read the Manual for the first steps, to lookup some stuff later on, and for the api reference.
- There are live demos available, which can be opened in the web browser without any preparation needed.
- It's a true **Free Software** project without commercial pro-versions.

1.3 Up-to-date?

Are you currently reading from another source than the homepage? Are you in doubt if that place is up-to-date? If yes, you should visit https://pseudopolis.eu/wiki/pino/projs/clove and check that. You are currently reading the manual for version 0.1.977.

1.4 Maturity

In this version, the state of clove is considered as production-stable.

1.5 Dependencies

There are external parts which are used by clove. Many thanks to the projects and all participants.

`banner image` *included* : _meta/homepage_bannerimage.png; public domain; copied from here.

`jquery` *included* : licensed under terms of GPLv2.

1.6 Introduction

Clove is a browser-side JavaScript library which mostly provides graphical widgets. In order to use Clove, you have to include it into your web page:

```
<!doctype html>
<html>
  <head>
    ...
    <link rel="stylesheet" href="clove.css" type="text/css">
    <script type="text/javascript" src="jquery.js"></script>
    <script type="text/javascript" src="clove.js"></script>
    ...
  </head>
  ...
</html>
```

The `jquery.js` can either be the included one or a different one, which is compatible. It contains the jQuery library, which is a 3rd-party component required by Clove.

1.7 First Steps

After you have included Clove in your web page, you can use its programming interface in some way in order to support your application.

The easiest and most straight-forward way is to start with an entirely empty web page (i.e. nothing in its `body`) and to include a script, which bootstraps your application frontend:

```
<!doctype html>
<html>
  <head>
    <link rel="stylesheet" href="clove.css" type="text/css">
    <script type="text/javascript" src="jquery.js"></script>
    <script type="text/javascript" src="clove.js"></script>
    <script type="text/javascript">
clove.build({
    view: "MainView",
    head1: "My first Clove application",
    body: {view: "Label", label: "Hello World!"},
});
    </script>
  </head>
  <body>
  </body>
</html>
```

The interesting part is the `clove.build` call, which constructs the user interface according to a 'blueprint'. The blueprint is a JavaScript object, which contains some key/value pairs, specifying details about the user interface. In this example we see a blueprint for a `MainView`, with some properties specified (including `body`, whose value is again a blueprint for an inner widget).

The following sections will explain more details about `clove.build` and the configuration values you can specify.

Please **Note**: The pattern of bundling many configuration values into one object - typically written as `{k1:v1, ...}` - and just using this object as a parameter is ubiquitous in Clove. It is not only used for widget configurations as here, but also at other places where many parameters are optional and/or vary depending on the context. **Whenever the documentation describes a parameter as a 'configuration object', or sometimes even just 'configuration', it is about such an object.** The documentation explains all relevant details about such objects for the affected functions. *This is something like 'optional parameters' for JavaScript.*

1.8 How To Build User Interfaces

In the next parts, we will see how to build a user interface in some more depth and how to use it for actual user interactions.

1.8.1 Building Widgets

As we have seen, user interfaces are built with the `clove.build` function. There are a few other functions, which also build user interfaces; in an equal way, but adapted for some specific situations. Directly constructing widgets by call the class constructor is not allowed. The `clove.build` call has the following general form:

```
clove.build({...}, {...});
```

The only two parameters it has are configuration objects (as mentioned above). The first one is a widget configuration (called 'blueprint' in the beginning). The second one is a build configuration, which controls some construction aspects. It is optional and not interesting in the beginning, while the first one is essential.

```
clove.build({view: 'Label'});
```

This is a very small widget configuration. Each widget configuration at least contains a `view` parameter, which specifies the widget class to instantiate. So this declaration controls what kind of widget to build.

Please **Note**: Although this is true enough for the moment, we will also see widget configurations without a `view` parameter later on. This is just because it can be implicitly clear in some situations.

Depending on the widget class, several other parameters exist. All of them are optional, but some are essential for making any real use of the widget. The `Label` widget shows just a piece of text; the `label` parameter of it specifies this text:

```
clove.build({view: "Label", label: "Hello World!"});
```

A few parameters are available for all widgets, as we will see later on, while many others come from the specific classes. Whenever you are interested in the properties available for a particular class, or whenever you want to learn more about any other part of the programming interface, read the relevant parts in the Developer Documentation.

For an overview of the existing widget classes, please see the *WidgetShowroom* demo. You can find the demos in your package, e.g. `clove/_meta/demos/WidgetShowroom/index.html`, or on the Clove homepage.

Please **Note**: The entire Clove programming interface resides in `clove.`, so the complete way to refer e.g. to the label class is `clove.Label`. The `view` parameter allows to omit the `clove.` though.

Some widgets are containers, which host one or more inner widgets. Those inner widgets are specified in configuration parameters as well, so a natural nesting results:

```
clove.build({
    view: "MainView",
    head1: "My first Clove application",
    body: {
        view: "Label",
        label: "Hello World!",
    },
});
```

This builds a widget of class `MainView`. This class can host one inner widget specified in `body`. This nesting can go arbitrarily deep and is used to structure complete user interfaces in one call.

clove.populateUI

This section closes with the introduction of a tool, which is not required but recommended to use. The clove.↩ populateUI function enwraps time-consuming actions, as building a complex user interface can be, and gives the user some better feedback.

You should enwrap a complex clove.build call with it this way:

```
clove.populateUI(function(){
    clove.build({
        view: ...
    });
});
```

1.8.2 Layout And Alignment

Building complex user interfaces is possible with container widgets. Technically they are just normal widgets, but they have properties (i.e. configuration keys in a widget configuration) for child widgets. Some of them are rather special-purpose, providing some real functionality as well. Others are just intended for grouping other child widgets together and align them in some defined way.

The latter group is usually called 'layouts'. There are a few layout types, all implementing a different but general-purpose behavior of alignment for its child widgets.

1.8.2.1 Stack Layout

A stack layout can be either horizontal or vertical. A horizontal layout aligns its child widgets column-based, side-by-side, all with the full height. A vertical one aligns row-based, all with the full width respectively. They are configured this way:

```
clove.build({
    view: "VerticalStack",
    rows: [
        {view: "Label", label: "Hello"},
        {view: "Label", label: "... my friend!"},
    ],
});
```

A horizontal layout is implemented by `HorizontalStack` and uses the `cols` property for its childs.

Please **Note**: There is a shortcut for horizontal and vertical stacks, which is commonly used: The `view` parameter may be omitted.

A larger example:

```
clove.build({
    rows: [
        {cols: [
            {view: "Label", label: "a1"},
            {view: "Label", label: "b1"},
            {view: "Label", label: "c1"},
        ]},
        {cols: [
            {view: "Label", label: "a2"},
            {view: "Label", label: "b2"},
            {view: "Label", label: "c2"},
        ]},
        {cols: [
            {view: "Label", label: "a3"},
            {view: "Label", label: "b3"},
            {view: "Label", label: "c3"},
        ]},
    ],
});
```

See also the *HeinersAsiaShop* demo.

1.8.2.2 Grid Layout

The last example tries to realize a 3x3 grid of labels. But as a grid it would not work great, because the column widths aren't connected. A real grid helps out here:

```
clove.build({
    view: "Grid",
    children: [
        {view: "Label", label: "a1", row: 0, col: 0},
        {view: "Label", label: "b1", row: 0, col: 1},
        {view: "Label", label: "c1", row: 0, col: 2},
        {view: "Label", label: "a2", row: 1, col: 0},
        {view: "Label", label: "b2", row: 1, col: 1},
        {view: "Label", label: "c2", row: 1, col: 2},
        {view: "Label", label: "a3", row: 2, col: 0},
        {view: "Label", label: "b3", row: 2, col: 1},
        {view: "Label", label: "c3", row: 2, col: 2},
    ],
});
```

1.8.2.3 Finetuning

There are some ways to finetune the alignment within such layouts. They make sense isolated or in some combinations in many situations.

Stretch Affinities

A layout has to divide the available space on one or both axes. In the first place this is done by asking the widgets for a preferred size. If more space is available, it can distribute it among the childs in a configurable way.

Assigning numbers to the childs' `horizontalStretchAffinity` and/or `verticalStretchAffinity` properties will lead to a distribution of free space in this exact proportions.

```
clove.build({
    cols: [
        {view: "Label", label: "Foo", horizontalStretchAffinity: 2},
        {view: "Label", label: "Bar", horizontalStretchAffinity: 1},
        {view: "Label", label: "Baz", /*horizontalStretchAffinity: 0*/},
    ],
});
```

Custom Sizes

The preferred size of child widgets is not the right choice in each situation. Sometimes it is required to set something fixed (and e.g. use scroll views if needed) in order to make the composition work. For such cases, specify `width` and/or `height` for some childs (as css length).

```
clove.build({
    cols: [
        {view: "Label", label: "Foo", width: "50pt"},
        {view: "SomethingDifferent", ..., height: "50pt"},
        {view: "Label", label: "Bar", width: "50pt"},
    ],
});
```

Stretching

Typically a widget uses all the space it can get from the layout. It bids for getting additional space by means of the `horizontalStretchAffinity`/`verticalStretchAffinity` properties. Even if it is set to 0, there is no guarantee to not get additional space.

The `strictHorizontalSizing` and `strictVerticalSizing` properties control if a widget is just placed within that space or if it actually fills the additional size.

```
clove.build({
    cols: [
        {view: "Label", label: "Foo", strictVerticalSizing: true},
        /* ... */
    ],
});
```

1.9 Widget Names

After constructing a user interface, you probably will need to access some of the widgets for working with its content or state. Since you specified just the blueprint, you don't have any direct access to the constructed widgets.

The easiest way is to assign a `name` to interesting widgets and to get the actual widget instances by those names afterwards:

```
clove.build({
    cols: [
        {view: "Label", label: "Foo", name: "foolabel"},
        {view: "Label", label: "Bar", name: "barlabel"},
    ],
});
```

After executing this build call, use the clove.getByName method for getting the widget instances:

```
var foolabel = clove.getByName("foolabel");
// do something with it ...
foolabel.setLabel("Fuh");
```

1.9.1 Name Scopes

The model behind widget naming is called 'name scopes'. They are indeed a bit more complex that just `getBy↩Name`. There can be many separated partitions of names. This avoids name conflicts between different contexts. On the other hand, a namescope can have child scopes, which are automatically included in name lookups.

When it comes to #h_customwidgets we will learn more, but for the moment we can use namescopes whenever we build multiple Clove user interface and want to avoid name conflicts. This is an advanced topic you can also skip at first.

For a new namescope, just construct `clove.RootNameScope` and use it in the build configuration (this is the second parameter of clove.build):

```
var mynamescope = clove.RootNameScope();
clove.build({
    rows: ...
}, {
    nameScope: mynamescope,
});
```

Each name used in the widget configuration is now known in `mynamescope`. If you would execute such code at two different places, you would end up with two isolated namescopes and no conflicts.

Getting the widgets is now possible with the namescope object:

```
var foolabel = mynamescope.getByName("foolabel");
```

1.10 Common Widget Functionality

The common base class for all widgets is `clove.Widget`. It implements some common functionality which is automatically available for each widget. All of its subclasses can be build and configured as introduced above. A later chapter will also explain how to implement Custom Widgets.

1.10.1 Widget Property System

The most essential thing all widgets have in common is the property system. It provides the foundation for all ways of working with the widgets' contents and states.

The root of this system is a key/value map of property values. The key is a simple description string like `'label'`, the value can be an arbitrary JavaScript object. This map is used for the complete configuration and state information of a widget instance.

You can access them by clove.Widget.getProperty and clove.Widget.setProperty:

```
var foolabel = clove.getByName("foolabel");
foolabel.setProperty("label", "Fuh");
```

While you can set values for arbitrary keys, some of them will be understood by the widget implementation and processed in some way. The `label` property is understood by a clove.Label, so the last example actually makes sense.

Those properties are the same as the one you set in a widget configuration, so it's the same `label` property as we already have seen in former examples (analogously you could overwrite the `rows` property of a `VerticalStack` with new child configurations) - but this way it is possible to dynamically change and retrieve those values.

For the well-known properties of a class, there is always a dedicated getter and setter as well:

```
var s = foolabel.label() + "xyz";
foolabel.setLabel(s);
```

1.10.2 Common Widget Properties

On top of this property system, the clove.Widget class provides some common properties (see the Developer Documentation for details):

- `visibility`: The visibility of a widget.
- `enabled`: If a widget is enabled, i.e. can interact with the user.
- `styleClass`: Additional css classes for styling. See #h_styling for more.

1.10.3 Common Widget Management

There are also some common management methods, i.e. for removing widgets. See the Developer Documentation for them.

1.11 Events

The Clove event system is mostly built by the clove.Event class. It implements an event, which can be triggered from one party in certain situations and can be listened to by other parties. Events can be arbitrary things, but often have to do with user interaction like a mouse click on a certain widget.

Although not technically, there is a conceptional distinction between the owner of an event (could be a certain instance of clove.Button) and the listeners (the program components which want to react on this event). They use a clove.Event instance in different ways.

Typically event names begin with 'On' like in `OnClicked`.

1.11.1 Handling An Event

Any party can listen to a certain event instance in order to react on it. It needs to have access to the event instance and to call clove.Event.addHandler on it:

```
mybutton.OnClicked.addHandler(function(e){
    // do something
});
```

The `e` parameter is a clove.EventArgs instance, which provides some execution control and holds event-specific information.

Alternatively to this way, you can directly specify an event handler in the widget configuration:

```
clove.build({view: "Button", OnClicked: function(e){...}});
```

Please **Note**: Call clove.Event.removeHandler for unregistering the handler (or see clove.Event.addHandler for other ways).

1.11.2 Triggering An Event

The owner of an event triggers it when a particular situation occurs. At first it has to construct it and make it available to potential listeners somehow.

```
this.OnSomethingGreatHappened = new clove.Event();
```

Note that there is one instance per owner and that the coupling between them is loose.

The owner triggers the event with some additional information, which leads to the execution of all handlers:

```
this.OnSomethingGreatHappened.trigger({answer: 42});
```

1.12 Datasources

Datasources are an abstraction for how to get and modify a certain data store (either a real one or a virtual one computing live values). Mostly for structured views - as tables, trees, ... - they provide the actual sources of data for displaying.

```
var mydatasource = new MyLottoNumberPredictionDatasource();
clove.build({view: "TableView", datasource: mydatasource});
```

The abstract base class for each datasource is clove.Datasource.

Before going deeper into the model, a ready-to-use implementation is introduced.

See also the demos about datasources.

1.12.1 Native JavaScript Datasources

While the datasource model is flexible enough for adapting it to arbitrary sources, there is one very simple implementation of it, which can directly be used and populated from outside. Just construct a clove.NativeDatasource instance and fill it with some data, then assign it to some view.

```
var mydatasource = new clove.NativeDatasource();
mydatasource.root().addRow(["Onion Marmelade", "10€"]);
mydatasource.root().addRow(["Rat Juice", "6.50€"]);
clove.build({view: "TableView", datasource: mydatasource});
```

The interface provides some more. Please find more details in the Developer Documentation.

1.12.2 Datasources Model

As mentioned before, clove.NativeDatasource is just one implementation of the base class clove.Datasource. You can implement custom subclasses and use them instead. This allows you to take the data not just from a JavaScript object, but from virtually everywhere.

In general, the data model is the following:

- There is one root node.
- Each node has a table like form with rows and columns building cells.
- Each cell has a data value; typically a String or number.
- Each cell has also is a new node.

The first three points are exactly what you would see in the spreadsheet tool of your favorite office suite, at least if you don't think too deeply what 'root node' should mean (maybe the root node is the worksheet in that analogy).

The last point might make it more difficult to imagine this construct graphically. Each cell in this table does not only have a value, but also is one more table. This can be nested arbitrarily deep.

Please **Note**: Every single node in this structure also is a table (it just could be an empty one), while each table cell is a node! Thinking ahead, this means 'node', 'table' and 'table cell' are interchangeable; which can be a confusing thing at first.

Typically you will not need the full flexibility of this model. However, the typical figures can easily be represented in such a model:

- Scalar values: Store your value in the root node.
- Lists: Store your values in the first column of the root node.
- Tables: Store your values in the rows and columns of the root node.
- Trees: Store each first-level nodes in the first column of the root node (as for lists). Then, for each of those nodes, go to the associated cell and insert the direct childs in the same way in its subtable. Do this recursively for each node until the tree is complete.

The clove.Datasource base class provides an interface which directly reflects the data structure. This must be implemented by a custom subclass, so external data consumers are able to retrieve your data structure by it. The Developer Documentation gives a full interface overview, while the following explains how a datasource works:

- clove.Datasource.getValue(ptr): Returns the value in a certain cell specified somehow by `ptr`.

For the root node, a consumer uses `undefined` as `ptr`. How any inner node, it uses another datasource function to get a pointer to it:

- clove.Datasource.valuePointer(irow, icol, parent): Returns a clove.DatasourceValuePointer, pointing to a non-root node inside the datasource. It is specified by a row and column index and a parent node. If the parent node is not the root node, you would use the same method for getting a pointer to it. This nested calls directly reflect the nested data structure.

Example:

```
mydatasource.getValue(
    mydatasource.valuePointer(1, 3,
        mydatasource.valuePointer(3, 7)
    )
);
```

This returns the value of cell 1/3 of the node in cell 3/7 of the root node.

1.12.2.1 Implementing A Custom Datasource

The already mentioned methods and some more must be implemented in a datasource. The first steps are again about the clove.DatasourceValuePointer class.

- clove.Datasource.valuePointer(irow, icol, parent): This method constructs and returns such a pointer:

```
valuePointer(irow, icol, parent) {
    //...
    return new clove.DatasourceValuePointer(irow, icol, backendObject);
}
```

The major trick here is the `backendObject`. A backend object for a node is an object, which is opaque for Clove and for consumers, but which contains everything your custom datasource implementation needs to answer data queries to that node. This could be custom index structures, references, or direct parts of your custom data structure.

When this function is called with `parent:=undefined`, you have to use an internal representation of the `irow/icol` cell in the root as `backendObject`. Otherwise you have to use the representation of an inner structure. For finding the right path in your structure, you must take care of the backend object you can get from `parent:`

```
var myRootFoo = new Foo(); // our external source
//...

class MyDatasource extends clove.Datasource(Object) {

    valuePointer(irow, icol, parent) {
        if (parent)
            var parentfoo = parent.backendObject;
        else
            var parentfoo = myRootFoo;
        return new clove.DatasourceValuePointer(irow, icol, parentfoo.getInnerFoo(irow, icol));
    }

    //...
```

- clove.Datasource.getValue(ptr): Return the value for the specified node:

```
getValue(ptr) {
    return ptr.backendObject.getFooValue();
}
```

This alone should be enough to fetch data from an arbitrary place in your structure. There are some more parts required though:

- clove.Datasource.rowCount(parent) and clove.Datasource.columnCount(parent): For a given parent node, return how much rows and columns it has:

```
rowCount(parent) {
    return parent.backendObject.getInnerFooRowCount();
}

// same for columnCount
```

- clove.Datasource.parent(ptr): Return the parent for the node given as clove.DatasourceValuePointer `ptr`:

```
parent(ptr) {
    var pfoo = ptr.backendObject.getParentFoo();
    if (pfoo == myRootFoo)
        return undefined;
    else
        return new clove.DatasourceValuePointer(pfoo.rowindex(), pfoo.columnindex(), pfoo);
}
```

The last essential thing is to notify consumers whenever something changed in your data structure. Otherwise you would never - or only randomly - get updated views. It could also fail when it assumes some row or column counts while some of them disappeared meanwhile in your structure.

In order to correctly notify consumers, some events from your clove.Datasource instance must be triggered:

```
this.OnDataInsert.trigger({r1: ..., r2: ..., c1: ..., c2: ..., parent: ...});

this.OnDataRemove.trigger(/* as above */);

this.OnDataUpdate.trigger(/* as above */);
```

Those events are for creation, removal and updating of nodes. The parameters are `parent`, which is a pointer as described above, and first/last row and column index which is affected by the event.

1.12.3 Headersources

There is an additional concept which is about the headers of rows and columns. In order to configure them (mostly the header text), some views also accept a headersource. This is a different interface, potentially implemented by a different object, which provides those information for entire rows and columns. However, the interface works similar and very often both interface are implemented by the same object.

Please read about clove.Headersource for details.

Please **Note**: The clove.NativeDatasource implementation introduced above also implements this interface and also provides methods for configuring the headers from outside.

1.12.4 Data Bindings

While datasources are the only way to display data in clove.DataView widgets, they are also part of the powerful data binding foundation. This allows to bind a datasource to arbitrary properties of arbitrary widgets.

For binding a node in a clove.Datasource to a property of a widget, a clove.DataBinding is to be used. This should be created with the clove.databind function and

- either connected in a widget configuration:

```
clove.build({view: "EditBox", text: clove.databind({datasource: mydatasource})});
```

- or later on with clove.Widget.bindProperty:

```
mywidget.bindProperty("text", clove.databind({datasource: mydatasource}));
```

Read about clove.databind for more options.

1.12.5 Asynchronous Data Sources

Asynchronous data sources are implemented by clove.AsyncDatasource. There are some interesting subclasses like clove.AjaxAsyncDatasource. Read the api documentation for details.

1.13 Dialogs

A dialog is a body area with a title bar, floating above the rest of the interface (looks and feels like dialogs in desktop environments). Be aware that they are simulations which run inside the main browser window. Opening an actual popup or a new browser tab is a different story.

There are some ways to create dialogs, with different simplicity levels which of course also differ in power.

See also the *HeinersAsiaShop* demo.

1.13.1 Simple Dialogs

Message dialogs with clove.messageDialog can show a message text in a dialog and requests the user to press a button. This dialog is modal, so the interface behind is not reachable. When the user has clicked on a button, the dialog is closed and an event is triggered. Example:

```
var OnProceed = clove.messageDialog({
    message: "Hello World!",
    buttons: ["Foo", "Bar", "Baz"],
});
OnProceed.addHandler(function(e) {
    console.log("Clicked on #" + e.button);
});
```

A similar dialog with a input text box is provided by clove.inputDialog. A slightly more complicated variant with a custom inner part (but including the button bar) is provided by clove.conversationDialog. Read about them for details.

1.13.2 Complex Dialogs

The widget class clove.Dialog implements the container with the frame and title bar, used to be the dialog. Like other containers, it has a property for an inner widget. But the usage it typically different, because direct usage in clove.build would lead either to a full-screen dialog or to some dialog-like looking box within your main interface. Both is obviously not intended behavior for a dialog.

Instead, use clove.Dialog.show, mostly in the same way as clove.build, and build an arbitrary inner interface for this dialog:

```
var dlg = clove.Dialog.show({
    view: "Dialog",
    title: "My Dialog",
    body: {view: "Label", label: "Hello World!"},
});
```

Although not for clove.build, the return value of clove.Dialog.show is important to keep. Everything you build with this function is not part of the root namescope. Instead, you must use `dlg.namescope` for getting widgets by name.

The clove.Dialog widget itself is also available in this result as `dlg.widget`. This is particularly useful for eventually closing the dialog later on with `dlg.widget.close();`.

1.13.3 Popups

Even more generic is to use clove.utils.popup. This opens any kind of widget floating in the same way as a dialog would do. Read about it for details.

1.14 Internationalization

Internationalization of a user interface has many facets. Some of them should be handled fine directly by the browser (this is true e.g. for string representations for numbers, dates and times). String translation is the next big topic and is what Clove helps you with.

There is even more, like left-to-right vs. right-to-left text flow. But those are beyond the scope of this manual.

The Clove Internationalization support is implemented in the clove.I18N class. There exists a single instance of this class as `clove.i18n`.

The idea behind translating a user interface is simple:

- Collect all your interface strings in a table, translated into each target language:

```
clove.i18n.addString('HelloWorld', {en:"Hello world", de:"Hallo Welt", nl:"Hallo wereld"});
clove.i18n.addString('ThanksForVisiting', {en:"Thank you for v...
```

- Take strings from this object whenever a user interface text is build:

```
foolabel.setLabel( clove.i18n.strHelloWorld );
```

This automatically does the translation to the language set in the user's system settings.

1.15 Custom Widgets

For complex and dynamic interfaces it might be a good idea to encapsulate some isolated parts to a new custom widget. Another reason for a custom widget could be to implement some entirely new behavior or functionality.

A custom widget class ...

- either directly or indirectly inherits from clove.Widget
- provides a constructor with the same signature as clove.Widget
- overrides some of the methods, depending on what it should do
- often introduces new properties
- virtually always implements adding some content to the html dom tree and working with it

A rather minimalist widget implementation is the following:

```
class MyHelloWorldLabel extends clove.Widget {

    constructor(config, domnode) {
        super(config, domnode);
        // this.contentnode is our html dom node
        this.contentnode.textContent = "Hello World";
    }

    // optional
    doinit() {
        // initialization steps ...
    }

}
```

Once this class is defined, you can use it in widget configurations:

```
clove.build({view: "MyHelloWorldLabel"});
```

For all kinds of graphical representation, you should use css as often as you can and especially follow the Styling section.

The `config` object is the widget configuration like it is passed in at the clove.build call. However, on the way down the constructor chain, this object can (and typically will) get changed. clove.utils.applyDefaultsToConfig is a typical tool in this context.

1.15.1 Widget Properties

Instead of the static text from the example above, you often want to provide a way to let the widget consumer decide about content or behavior. The widget property system, which does exactly this, was already introducted above from the consumer perspective. For the widget developer, providing a property `myproperty` means the following:

- Calling `this.declareProperty("myproperty", mydefaultvalue)` in the constructor. The default value is optional.
- Optionally provide `myproperty_getter` and/or `myproperty_setter` for dynamically computing property data or for reacting on changes, e.g. by changing stuff in the html content.

1.15.2 Layouts And Sizing

Widgets have to live together on a more or less large area, where they get a position and size. With the size actually allocated for the widget, the internal routines then have to align the own content in a proper way. The custom widget has to play together with these mechanisms. This can happen more or less complicated, as the following sections describe.

But beforehand, please note: clove.Widget comes with an existing implementation for all this sizing matters, which works for some simpler cases of html content. It will however fail whenever the content does some internal positioning or uses block elements and in many other cases.

1.15.2.1 Using An Existing Layout Implementation

A very easy and powerful way to build new widgets is to compose it of existing widgets and use a layout internally, instead of manually generating new html content. Obviously this does only work if the existing widgets are enough to build your custom one.

Implement such a composed widget by adding one of the layout mixings to the superclass chain and directly give it a configuration:

```
class MyLabelComposition extends clove.StackLayout(clove.Widget) {

    constructor(config, domnode) {
        // add some more stuff to (a copy of) the config
        config = clove.utils.applyDefaultsToConfig(config, {
            orientation: "vertical",
            children: [
                {view: "Label", label: "Foo"},
                {view: "Label", label: "Bar"},
            ],
        });
        super(config, domnode);
    }

}
```

1.15.2.2 Provide Custom Sizing Support

If the custom widget manages real own html content, it typically has to implement the following methods for sizing support:

- clove.Widget.computeMinimalWidth, clove.Widget.computePreferredWidth, clove.Widget.computeMinimal↩ HeightForWidth, clove.Widget.computePreferredHeightForWidth: Return minimal and preferred sizes on both axes for the widget in its current state as pixel values.
- clove.Widget.doresize: Realign the internal stuff properly into the current actual widget size.

Whenever the some relevant parts of the widget state changed, it has to call clove.Widget.relayout in order to refresh its sizes.

1.15.2.3 Prevent Name Conflicts

When you implement a new custom widget by somehow composing it of existing ones, you typically assign names to some internal parts, so you can access and work with them internally later on. But there is a pitfall: Those names would conflict with other ones outside of the widgets, or even with a second instance of the same custom widget. This can be solved this way:

- Place your custom widget in a new namescope. This can be done by adding clove.NameScope to your inheritance chain, so you might end up with a superclass like `clove.NameScope(clove.Stack↩ Layout(clove.Widget))`. This leads to an isolated namescope for all your inner widgets.
- If your container isn't a container, you are done. Otherwise there is a new problem now: When a consumer uses the new custom widget, placing also widgets in this container, and tries to access those ones by name, it will fail. This is because those widgets aren't in the namescope you would expect, but they are in that isolated new one. The solution is to introduce one more namescope, just for this containing part, and add just this one as child namescope to the original one.

```
class MyContainerWidget extends clove.NameScope(clove.StackLayout(clove.Widget)) {

    constructor(config, domnode) {
        super(clove.utils.applyDefaultsToConfig(config, {
            children: [
                ...,
                {rows: [], name: "innercontainer", newNameScope: true},
            ],
        }), domnode);
        this.declareProperty("body", {rows: []});
    }

    body_setter(v) {
        this.getByName("innercontainer").setChildren([v]);
    }

    doinit() {
        super.doinit.call(this);
        this.containingNameScope().addChildNameScope(this.getByName("scroll"));
    }

}
```

1.15.3 Best Practices

There are some guidelines for developing custom widgets, which should be known.

Add a css class to the top node

There is a another html dom node in each widget, `this.topnode`, which encloses the actual content node. This has different technical purposes and is also typically used to assign a css class to, which represents the widget class. This allows to style the widget parts.

```
class MyFooWidget extends clove.Widget {

        constructor(config, domnode) {
            super(config, domnode);
            $(this.topnode).addClass("myfoowidget");
            //...
```

clove.utils.suspendResizing

When a program logic does large changes on some user interface parts, this can be time consuming. Think about adding a large bunch of data to a widget. Depending on how it is designed, you might end up with calling something like `somewidget.addData(something);` lots of times. Each of this call likely will trigger the recomputation of the widget sizing. Most of those computations - all but the last to be exact - have no value at all, but can take a considerable amount of time.

For large computations within your widget routines, you should temporarily suspend the resizing.

```
var xxx = clove.utils.suspendResizing();
try {
    //...
}
finally {
    clove.utils.resumeResizing(xxx);
}
```

1.16 Styling

Styling in Clove should be done entirely with css classes. There are only rare cases where directly assigning styles to elements is the better way. Since a Clove widget should be represented by a css class as well, styling is natural.

```
.myfoowidget {
    color: blue;
}
```

More classes can be added at some place of the dom node for supporting finer styles.

Also note the dom structure each widget has: parent > topnode > contentnode > content. The topnode is the root node of a widget. Within it, there is the content node and within it the actual content. For some css selectors this is important to know.

```
.myfoowidget > div > a {
    color: blue;
}
```

This would colour each `a` node directly in the content node of a MyFooWidget.

1.16.1 Clove Common Styles

There are some common style classes. They help for easily styling some stuff in default situations, like larger control appearance, important texts, errors texts or margins. It's not required to use them, but they avoid some typing and makes application styling easier to adapt.

For a list of existing common classes, inspect code samples or `clove.css` and find css class names beginning with `clovestyle_`. Those kinds of classes are particularly interesting:

- Text styling: `clovestyle_text_*`
- Panels (i.e. container areas for some content): `clovestyle_panel_*`
- Margins around arbitrary widgets: `clovestyle_*margin*`

1.17 Using Widgets In Existing Webpages

The typical way is to build widgets completely in a Clove context. However, it is also possible to build widgets into existing parts of an existing webpage. This way reuses your existing page but just adds some inner parts to it.

At first you have to get the html dom node which shall be populated by a widget. Then, use this in the build configuration of the clove.build call:

```
var mydiv = ...;
clove.build({
    rows: ...
}, {
    domnode: mydiv,
});
```

You might also set the clove.Widget.doStandaloneResizing property on such a widget in order to make it automatically resize itself.

1.18 Packages

Here are the packages with brief descriptions (if available):

1.19 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

1.20 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

1.21 runtest Namespace Reference

Classes

- class WebconsoleHTTPRequestHandler

Functions

- def startserver ()

Variables

- bool running = True
- passed = None
- details = None
- testrootdir = os.path.dirname(os.path.abspath(__file__))
- string testgroupdir = testrootdir + "/" + sys.argv[2]
- test = sys.argv[1]
- browser = sys.argv[2]
- def callbackurl = startserver()
- string testurl = "file:///" + testrootdir.replace("\\", "/") + "/" + test.replace("\\", "/")

1.21.1 Function Documentation

1.21.1.1 startserver()

```
def runtest.startserver ( )
```

1.21.2 Variable Documentation

1.21.2.1 browser

```
runtest.browser = sys.argv[2]
```

1.21.2.2 callbackurl

```
def runtest.callbackurl = startserver()
```

1.21.2.3 details

```
runtest.details = None
```

1.21.2.4 passed

```
runtest.passed = None
```

1.21.2.5 running

```
bool runtest.running = True
```

1.21.2.6 test

```
runtest.test = sys.argv[1]
```

1.21.2.7 testgroupdir

```
string runtest.testgroupdir = testrootdir + "/" + sys.argv[2]
```

1.21.2.8 testrootdir

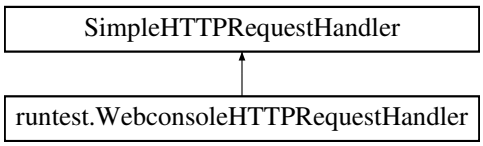
```
runtest.testrootdir = os.path.dirname(os.path.abspath(__file__))
```

1.21.2.9 testurl

```
string runtest.testurl = "file:///" + testrootdir.replace("\\", "/") + "/" + test.replace("\\",
"/")
```

1.22 runtest.WebconsoleHTTPRequestHandler Class Reference

Inheritance diagram for runtest.WebconsoleHTTPRequestHandler:



Public Member Functions

- def log_message (self, format, args)
- def do_HEAD (self)
- def do_POST (self)

1.22.1 Member Function Documentation

1.22.1.1 do_HEAD()

```
def runtest.WebconsoleHTTPRequestHandler.do_HEAD (
              self )
```

1.22.1.2 do_POST()

```
def runtest.WebconsoleHTTPRequestHandler.do_POST (
              self )
```

1.22.1.3 log_message()

```
def runtest.WebconsoleHTTPRequestHandler.log_message (
              self,
              format,
              args )
```

The documentation for this class was generated from the following file:

- _meta/test/runtest.py

Index