VAADIN TRAINING

Basics

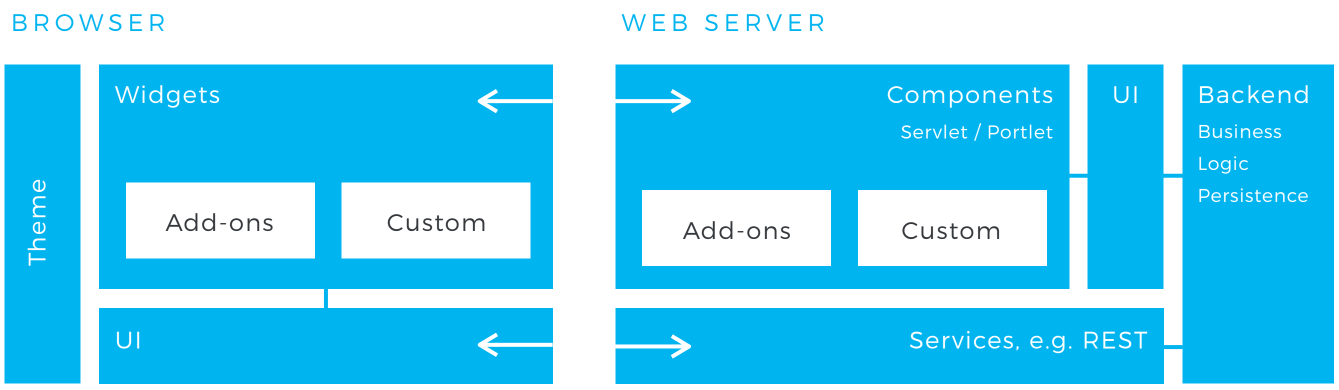
vaadin}>

Architecture

Vaadin is a server-side component framework. It means that the code you write is executed as Java on the server.

The client-side logic of the framework is based on using Google Web Toolkit (GWT). All state changes are managed on the server-side, and the client side merely reflects those changes in the browser.

Vaadin is event driven; your logic runs because the user interacted with a Component. The framework handles client-server communication for you.



- Vaadin widgets: Logic of how an individual component behaves in the browser
- Add-ons: You can extend the core framework with 3rd party widgets
- Custom: You can have your own custom widgets
- Theme: The theme defines the look and feel for your application. You may use an existing one or create your own
- UI: You can have Vaadin independent UI code in your browser. For example, your Vaadin application can be embedded into a web page

- Components: Server-side APIs of the components
- Add-ons: The server-side APIs for your 3rd party add-ons
- Custom: APIs for your custom widgets
- UI: The user interface logic of your application
- Backend: The business logic of your application
- Services: Web services that might be used by the Vaadin independent parts of your application

NOTE When using the server-driven programming model, your user interface logic and business logic only exist on the server and are never exposed to the client (browser).

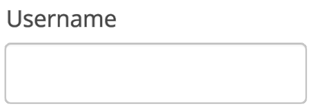
Components

Common Features in All Components

You can define a **caption** for all Vaadin components.

JAVA

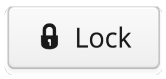
```
TextField textField = new TextField();
textField.setCaption("Username");
```



Icons can be assigned to components. Icons are in some cases rendered as a part of the component (Button) or in combination with the component (ComboBox). Icons are often Font Icons, but can be any image.

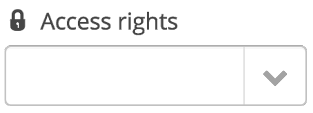
JAVA

```
Button button = new Button(“Lock");
button.setIcon(VaadinIcons.LOCK);
```



JAVA

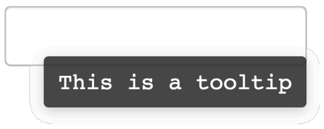
```
ComboBox comboBox =
        new ComboBox("Access rights");
comboBox.setIcon(new ThemeResource("lock.gif"));
```



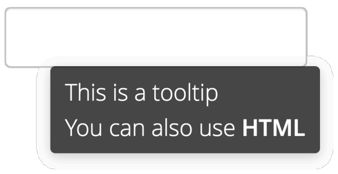
Defining a **Description** for a component will show the string as the component's tooltip.

JAVA

```
TextField toolTipField = new TextField();
toolTipField.setDescription("This is a
tooltip");
```



```
toolTipField.setDescription("This is a tooltip
<br /> You can also use <b>HTML</b>",
ContentMode.HTML);
```



You can toggle a component's visibility on the server side with **`setVisible()`**.

Non-visible components are not sent to the browser, but stay in the server-side state.

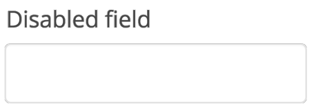
NOTE Changing the visibility of a component using CSS will still leave the component in the DOM tree.

JAVA
```java
Label invisibleLabel = new Label("This will not be visible");
invisibleLabel.setVisible(false);
```

You can control the availability of any component on the server-side by using the function **`setEnabled()`**.

- A disabled component will be rendered with less opacity than normal
- The server does not process events from disabled components

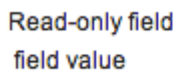
JAVA
```java
TextField disabledTextField =
        new TextField("Disabled field");
disabledTextField.setEnabled(false);
```



Component can be marked **read-only** with **`setReadOnly()`**.

- Read-only components do not accept new values, from client side

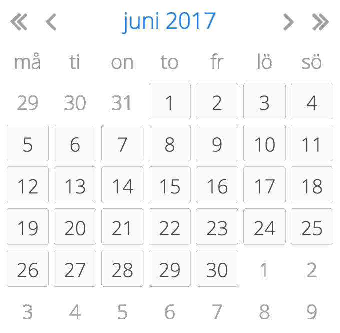
JAVA
```java
TextField readOnlyTextField =
        new TextField("Read-only field");
readOnlyTextField.setValue("field value");
readOnlyTextField.setReadOnly(true);
```



All components support **locales**. For example, by defining a locale for a DateField this will change the language in which the month and week names are rendered.

JAVA

```java
InlineDateField inlineDateField =
        new InlineDateField();
inlineDateField.setLocale(
        new Locale("sv", "SV"));
```


juni 2017

må	ti	on	to	fr	lö	sö
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	1	2
3	4	5	6	7	8	9

You can **define sizes** for components by using **`setWidth`** and **`setHeight`**.

- Sizes can be defined as relative (e.g. 100%) or as static (e.g. 200px)
- Relative sizes are relative to the parent component or a part of it (more info in Layouts course)
- If the size is undefined, then the component is rendered using its natural size
- **`setSizeFull() = setWidth("100%") + setHeight("100%")`**
- **`setSizeUndefined() = setWidth("-1") + setHeight("-1")`**

All components have an API for defining CSS class names.

- **`setStyleName()`**
  clears all previously added style names and adds the given style name
- **`addStyleName()`**
  adds a style name to the component
- **`removeStyleName()`**
  removes a given style name (if it exists) from the component
- See 'Theming your application' course for more information about styling

`getParent()`
returns the parent component (= component container) for any attached component.

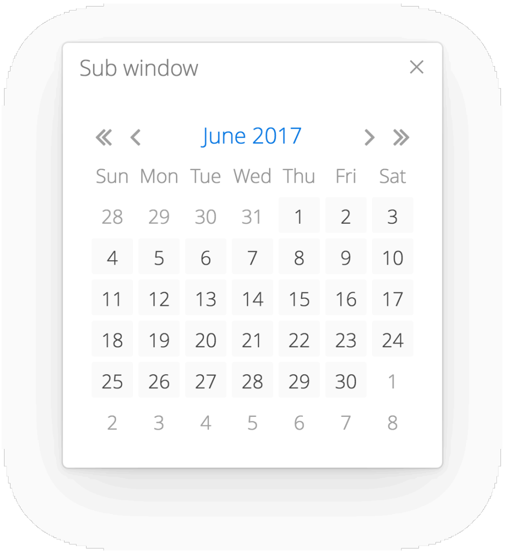
- If the component is not attached to a component container, **`getParent()`** will return null

`getUI()`
returns the UI instance to which the component is attached.

- If the component is not attached to a component container, **`getParent()`** will return **`null`**

`findAncestor(Class<T> parentType)`
returns the instance of the closest parent component with the given type.

- If a parent component with the given type cannot be found in the parent hierarchy, **`null`** is returned



`inlineDateField .getParent();`	→	verticalLayout
`inlineDateField .getParent() .getParent();`	→	subWindow

Check out the sampler to see component demos and code examples of how to use components
http://demo.vaadin.com/sampler