

A man with a beard and short hair, wearing a white shirt and a grey vest, is smiling and looking at a laptop. He is sitting at a desk with a tablet and some papers. In the background, another person is standing near a whiteboard in a bright office environment.

Less Code More UI

CaptainCasa Enterprise Client

Less Code

Less Code

Still many big application systems have failed to successfully move into modern, web-based application structures. And still many application projects suffer from a too high effort of web-based programming.

Lack of efficiency

The world of today's browser frameworks is frontend-driven. They are based on what the browser provides (HTML, JavaScript, CSS) and try to simplify and organize the development of frontends inside the browser.

This frontend complexity directly influences the efficiency of application front end development. A high level of skills is expected from developers, not only for developing screens but also for binding them efficiently to server-side processing.

Lack of long-term stability

The world of UI frameworks is a volatile one: there is constant coming and going of frameworks. Even big providers of frameworks are changing their frontend framework strategy constantly - setting the previous framework to deprecated and pushing the new framework as new hype.

For complex business applications this typically means some huge effort to adapt and to re-code.

CaptainCasa Enterprise Client

CaptainCasa Enterprise Client is a server-driven UI framework that was developed with the complexity of big business applications in mind.

The core problems "lack of efficiency" and "lack of long-term stability" are solved as part of the framework's architecture. The framework is open both on frontend side (integration of new components) and on backend side (integration to business logic frameworks).

It consists on the one hand out of a rendering client running as single page application within the browser. The rendering engine is following the so-called RISC method, which grants a maximum level of browser compatibility and that comes with an excellent performance.

The rendering client talks to the server processing through an http interface, in which layouts are sent from the server to the client and events / data changes are sent back to the server processing.

The server is a JAVA-EE based Java implementation that manages the application screens and that can smartly integrate into server-side Java frameworks to run the application logic.

Zero code on
browser side

Zero code on browser side

Reducing the amount of application code starts at client side - within the browser! CaptainCasa Enterprise Client completely unburdens the application developer from any coding within the frontend.

Zero code

No application development is done on client side - all application development is done on server side only.

This means:

- No JavaScript coding
- No HTML coding
- No CSS management

Zero fighting with HTML

Working with HTML technologies is exciting - but means to spend quite some effort in areas which are outside the application scope. With CaptainCasa Enterprise Client no efforts need to be spent in areas like:

- Browser incompatibilities
- Working with scripting languages which require much more quality attention than e.g. working with Java
- Technically managing different device sizes / device capabilities

Zero redundancy of logic

If coding inside the browser, then you pre-execute certain application logic on client side which is later executed on server-side anyway. This means that you have a certain degree of redundancy between the logic on client-side and the logic on server-side.

With the Zero-code-approach of CaptainCasa Enterprise Client all the logic is on server-side - there is no redundancy.

Zero HTML skill set expected

Application developers are not expected to provide any HTML skill set as consequence. A normal Java skill set is enough in order to quickly and efficiently develop screens from the server-side.

Zero split up of development

Application development teams are in many cases split up into the “frontend-guys” and the “backend-guys” - meaning high effort to synchronize both parts of the team.

With CaptainCasa Enterprise Client this separation is not required - any development is done in a consistent back end environment.

Zero browser-server
communication code

Zero browser-server communication code

The client processing within the browser needs to talk to the server-side processing. With CaptainCasa Enterprise Client this communication is completely covered by the framework.

Zero code

The communication between browser and server is part of the framework and is completely transparent for the application development.

Zero round-trip management

Binding a client-side processing to server-side logic typically means to carefully manage the communication round-trips in between. The number of round-trips and the data volume of each round-trip are essential for the application runtime performance.

Within the CaptainCasa framework a blocked data communication is guaranteed: each interaction of the user which is relevant for synchronizing with the server (e.g. pressing a button) leads into exactly one round-trip to the server-side. The data volume is restricted to what is visible within the current screen.

All grid data is managed in a way, that only the visible items are communicated from the server to the client. A grid may have thousands of items on server-side, but only

these items are communicated to the client that are currently visible.

Lightweight round-trip concept

All communication between client and server is based on the exchange of changes only. This means: if a screen is updated by the application interaction logic (e.g. switching from one content to the next), then only a reduced set of data is sent from the server to the client. This on the one hand reduces the data volume of communication significantly and on the other hand improves the speed of the client-side rendering.

Round-trips between the browser and the client are lightweight as consequence.

Zero exposing of fine-granular APIs

There is only one API between the browser and the server: the server sends layout definitions to the browser - the browser sends data changes and user interface events to the server side.

There is no need to expose fine granular functional server APIs for any part of the application to just serve the API requirements of the screen processing.

Less interaction
code on server side

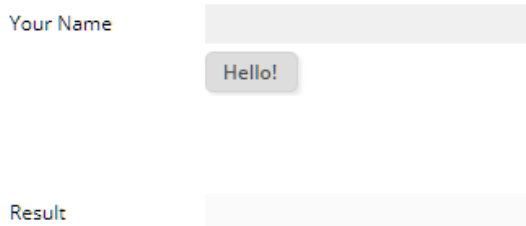
Less interaction code on server side

“Descriptive first” is the way to develop screens with CaptainCasa Enterprise Client. There is no code required to define which component needs to be placed at which places.

Zero layout code

The layout definition itself is not programmed but defined in XML. Re-positioning of components inside the layout, changing texts/images/etc. - all this does not affect any code.

Example: the following screen...



The screenshot shows a simple web form. At the top left, there is a label 'Your Name' followed by a text input field. Below the input field is a button labeled 'Hello!'. At the bottom left, there is a label 'Result' followed by a text area for the output.

... is defined by the following layout definition:

```
<t:rowdemobodypane rowdistance="5" >
  <t:row>
    <t:label
      text="Your Name" width="120" />
    <t:field id="g_ccpreview_4"
      text="#{d.Helloworld.name}"
      width="200" />
  </t:row>
  <t:row>
    <t:coldistance width="120" />
    <t:button
      actionListener="#{d.Helloworld.onHello}"
      text="Hello!" />
  </t:row>
  <t:rowdistance height="50" />
  <t:row>
    <t:label text="Result"
      width="120" />
    <t:field enabled="false"
      text="#{d.Helloworld.output}"
      width="100%" />
  </t:row>
</t:rowdemobodypane>
```

This layout definition can be created by using a WYSIWYG layout editor tool.

Less interaction code

Finally, there is some code - for processing data changes and events from the front-end side and for managing the interaction.

```
public class Helloworld
{
    String m_name;
    String m_output;

    public void setName(String value)
    { m_name = value; }
    public String getName()
    { return m_name; }

    public String getOutput()
    { return m_output; }

    public void onHello(ActionEvent ae)
    {
        if (m_name == null)
            m_output = "No name set.";
        else
            m_output = "Hello world,"
                +m_name+"!";
    }
}
```

The code represents the so-called View-Model of the corresponding screen. - Each screen definition (XML) and its interaction (Java) form a unit that is encapsulated via interface and events, so that it can be easily embedded and re-used by any other screen.

Dynamic layout

Of course, not all screens are statically defined. Example: you may want to automatically create a screen out of some meta data, e.g. of a business object definition.

The XML that is normally passed by a static file-definition now is internally passed as node-hierarchy - that's the only difference. Still the layout and arrangement of controls is separated from the interaction code behind!

Less binding code on
server side

Less binding code on server side

From application developer's point of view CaptainCasa Enterprise Client is the "landing zone" of the UI on the server-side. Each screen or part of a screen is reflected by some corresponding object instance within the server at runtime.

How this object instance now works with the business logic below, this is up to the concrete usage scenario. Due to the usage of Java and open standards there is no problem to bind to any framework.

Direct binding to business logic

There are many ways to structure the business logic on server side within Java. Examples:

- Spring
- EJB
- Direct Java-implementation based on Hibernate / JPA

All these ways somehow allow to pass data into and out of the logical processing. The typical building blocks are:

- Data Objects to represent the data structure - typically implemented as so-called "PoJo"-objects (plain old Java objects) or implemented as hash-table-data-holders.
- Object Factories to keep objects and to load objects.
- Transactions.

When implementing screens using the CaptainCasa framework then the interaction layer can flexibly adapt to the logical layer - re-using all structures that are exposed.

Example:

- Data structures of the business logic layer can be directly referenced from screens.

Composite controls

Any screen or part of a screen that is created within CaptainCasa Enterprise Client can be re-used in other screens - e.g. as part of the other screen or as popup-screen. Each screen is represented by one object instance and provides an explicit interface to initialize, to update and to listen to screen events.

An implementation of a screen is a "composite control" by itself and may be used for certain purposes:

- Encapsulation of "high value components" that are re-used many times. Example: an address-input-pane
- Providing of configure-able components to take over complex parts centrally. Example: a generic grid which is bound to a list of data objects

There is no difference between "developing a screen" and "developing a composite component".

Rapid prototyping

For rapid prototyping the UI processing can be simply built without any serious business logic behind, but with some simulation of business logic.

All calls from the user interface to the logical layer are directed to a so-called facade-interface which is implemented by some dummy logic at the beginning. - Later, the real logic can be plugged behind step by step.

More UI

More UI

The world of web user interfaces is confusing: on the one hand there are millions of great web pages available throughout the world - on the other hand the development of complex web pages still is not simple at all and there are important issues which are still not properly solved with HTML based technologies.

CaptainCasa Enterprise Client overcomes limitations of the browser by using a rendering concept that follows the RISC method. All components are built on a set of basic, simple browser elements which are such basic, that they are supported by any

browser. Complex components are assembled out of these basic elements.

This rendering concept not only defines a unique level of browser compatibility, but also is the technical reason for decoupling the functional capabilities of components from the browser's capabilities.

An own rendering layer that is solid, fast and stable is the starting point of any front-end framework. And it is the central point of independence in the volatile world of web frameworks.

Rock solid, high
performance
browser processing

Rock solid, high performance browser processing

CaptainCasa follows the so-called RISC method on browser side: only a small subset of basic HTML elements is used. All functional components (button, field, combo box, grid, menu, ...) are assembled out of these basic HTML elements. The assembly is done within a lightweight JavaScript library.

Unique browser compatibility

This approach guarantees a unique compatibility throughout various browsers. All the basic elements and the way they are used are supported by any browser.

All access to these basic elements within the client is shrink-wrapped by some JavaScript “kernel layer”, which rules the access to the basic elements and in which - if any - browser dependencies are managed. This kernel layer is small - and only consists out of 2.500 lines of JavaScript code.

The assembly level (JavaScript classes) on top is simple JavaScript processing as well.

Excellent performance

The browser is used in a very performant way. The browser's role is to render simple elements (e.g. DIV elements) at defined coordinates only.

Layouting (i.e. the decision where to place which component following dedicated rules) is executed within corresponding components on a JavaScript level. - JavaScript in the meantime is a language which is executed with high performance, utilizing just-in-time compilers.

Rock solid

There are no dependencies to other HTML frameworks when it comes to the rendering. There are only limited dependencies to HTML due to the RISC method that is used. The layouting (i.e. the concrete positioning of controls) is done by layout management components (and not by the browser).

Consequence: the rendering processing is rock solid and predictable.

Overcoming browser limitations

Overcoming browser limitations

Browser layouting is limited in many areas

Still the browser has layout limitations which have their reason in the fact that its history comes from rendering text data. Example: vertical layouting in the browser is done with the strategy of rendering a page with some infinite height.

Pretty normal situations like “fix header - scrollable body - fix footer” are already causing problems with the browser’s default layouting and as consequence require some explicit JavaScript management. This gets even more complex if pages/screens are nested one into another.

Overcoming the limitations

Within the CaptainCasa framework all components are based on a screen based rendering concept instead of a page-based rendering concept.

All layout managers are mini-JavaScript programs that can follow any layouting strategy - there are no limits.

The default component library comes with sophisticated layout managers that cover both the requirements of...

- general screen layouting (Example: header, scrollable body, footer)
- complex form processing (mixture of components, some defined with percentage sizes, some with pixel sizes)
- graphical processing (x,y,z positioning)
- adaptive layouting (layout adapts to available physical space)

One place of layouting

In normal web developments the layout is defined at several places - especially if it comes to adaptive layouts:

- The HTML elements that are created
- The CSS definitions that e.g. define media-dependent layouts
- JavaScript processing that is somewhere in between.

With CaptainCasa the layouting is defined exactly at one place: within the component that manages the layout. This makes it very easy to...

- exactly follow why the layout is assembled in a certain way
- embed one component into another, without mixing and mashing layouting strategies

Long term browser stability

Long term browser stability

The life-cycle of a business application typically exceeds the life-cycle of today's web frameworks.

CaptainCasa Enterprise Client in its history proved that it is possible to have one user interface architecture, which spans the life cycle of your application.

Independent HTML stack

CaptainCasa Enterprise Client comes with its own HTML stack for two reasons:

The principle to use low level elements only as base for complex components ensures the browser compatibility and flexibility that is used to long term serve the UI requirements of business applications. The CaptainCasa HTML stack is no encapsulation of any existing HTML technology but is a way of managing HTML in an efficient, flexible and robust way.

The market of HTML frameworks is a volatile one, in which frameworks constantly are hyped on the one hand and are deprecated on the other hand. The size of these framework-stacks typically is a huge one and the investment to use it is huge as well. - As consequence the deprecation of an HTML framework stack has massive consequences.

By using some own HTML stack, CaptainCasa is independent from any other framework - both from functional point of view and from volatility point of view.

Explicitly decoupled client

The thin client principle of the CaptainCasa framework ensures that the actual UI

implementation (i.e. the rendering of controls) is completely decoupled from the application's screen implementation (i.e. the interaction logic on server side).

The client is some independent program - in the browser a single page application built with JavaScript - to render screens according to a protocol. In principle the whole client can be exchanged from one technology to the next without even telling the application screen processing about.

CaptainCasa history: from Swing via JavaFX to RISC-HTML

Exactly this happened two times within the history of CaptainCasa:

- CaptainCasa started with a Java-Swing-based client in 2007
- A parallel, compatible client was provided using Java FX in 2013
- A web client ("RISC Client") was provided in 2017.

All clients are based on the same protocol between client and server and all clients share the same control library. As consequence there was only minimal ("zero") effort to transfer a former Swing-based application into a web-based application by just exchanging the rendering client - without applying changes to the server side application.

Rich component library

Rich component library

The size and quality of the component library is a core issue when starting application projects. Even though the idea of assembling different components from different sources into one screen might technically sound nice, it is important to start with a rich set of consistent components.

Rich set of components

The CaptainCasa component library includes more than 100 components. They can be separated into:

- Dialog-window components (modal, modeless dialog-windows, standard dialog-windows)
- Structure components (header, body, footer, status bar, ...)
- Layout & Container components (pane, tabbed pane, scroll pane, row, overlay area, adaptive area, ...)
- Data components (label, field, combo box, combo field, slider, checkbox, radio button, ...)
- Animation components
- Grid and tree components
- Integration components (charting, Google Maps, Open Street Map)
- Components for WYSIWYG editing of content
- Invisible components (timer, reaction on server-side events, client data-exchange, ...)

Consistent set of components

All components share a consistent way of sizing, rendering, styling, event management,

support of keyboard, drag & drop and management of popup menus.

High level of quality within components

The set of components and the properties to control the components were defined from 2007 on and constantly improved. As consequence there is a high level of built-in functions, that simplify the creation of typical application screens.

Example: the default FIELD component includes:

- Definition of data type to be checked during input; checking against regular expressions; max length, lowercase, uppercase support
- Support of internationalized data types (date, time, decimal)
- Sophisticated ways of helping the user (from classical F1-help to user hint popups, interactively showing up)
- Different ways of triggering events to the server side (e.g. triggering server round-trip after n milliseconds of inactivity)
- Sophisticated background drawing (error field, mandatory field)
- Sophisticated focus management, explicit setting of focus, individual tab-order
- Possibility to add any type of value help dialog-windows

Integration of new graphical components

The integration of new components is done through a simple set of JavaScript interfaces and by some registration within server-side xml-files. There is a simple integration layer for integrating other pages (e.g. Google Maps, charting libraries, ...).

Consistent
application
look & feel

Consistent application look & feel

Hundreds of screens...

A business application typically provides some hundreds of screens to the user. Many of them are “boring” (customizing screens, master data screens), some of them are really eye catching (planning boards, interactive charts, ...).

Governing the access to UI features

CaptainCasa limits the direct access to native browser UI features for the application developer - and governs the access by making each new feature part of the central control library, so that everyone can benefit.

Typically, the implementation of new UI features is done by some client experts which are familiar with JavaScript / CSS / HTML. The features themselves then are brought to the application developer as new control or as extended attributes of existing controls.

Simple definition of style and style variants

The style management is simplified by defining the style as XML and editing the style by using some comfortable tool. The CSS-file is generated from the XML definition. The XML definition is based on variables for central values (e.g. font family, central color definitions), so customer-specific styles can be easily created both at design- and at runtime on a variable level - without knowing anything about CSS.

For each component (e.g. button) you can define both a default style and dedicated style variants. When placing a component into a layout you define which style variant to use. As consequence you can set up central style variants for all controls, that represent a certain usage of the control (e.g. button is used in the header area).

The style management includes a simple to use tool for end-users to define own styles at runtime.

Composite Components

Each screen that you define with CaptainCasa is a re-useable artefact. This means: you may e.g. define a central “Address pane” and re-use this composite component everywhere in your application where address in/output is required.

Composite components on the one hand provide the screen layout (arrangement of e.g. labels and fields), on the other hand they define the interaction that is processed within the screen.

Composite components can be “very concrete” (e.g. address-pane) or “very dynamic” (e.g. abstract grid view to render an array of data).

Page templates

You can set up own page templates so that the general structure of a page / part of a page is following some template.

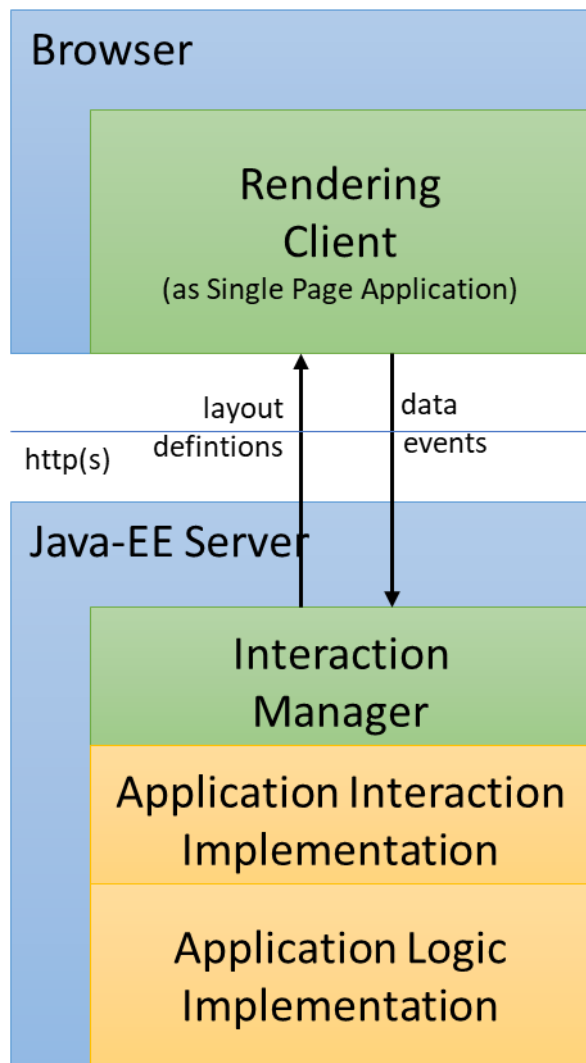
CaptainCasa

Enterprise Client

Captain Casa Enterprise Client in a nutshell

High level architecture

CaptainCasa Enterprise Client is following the so called “Thin Client Architecture”:



The word “Thin Client” is not at all referring to the graphical capabilities of the rendering client! - But is referring to its role in the architecture. The client is a pure rendering engine which renders layout definitions that are sent from the server side.

The rendering client does not know the business semantics behind the layout (e.g. if it represents material master date or a purchase order). The client just sees the

layout which comes as XML definition and renders it into corresponding controls.

On server side an interaction manager is the one to send the layouts to the client and to receive data changes and events from client side.

The interaction manager is connecting the screen processing to the application interaction layer.

Client architecture

The client is a JavaScript program running as single page application within the browser.

It uses a JavaScript control library which comes from CaptainCasa itself. This control library is based on the so-called RISC method. This means that HTML is used on a very basic level only - ensuring a unique browser compatibility on the one and a great control flexibility on the other hand.

Server architecture

The server architecture is internally using JSF (Java Server Faces). JSF is the Java-EE standard for server-side interaction management - and is typically used in classical HTML scenarios, in which there is a constant sending of complete pages from the server to the browser.

JSF is an open standard: within CaptainCasa Enterprise Client it creates the XML-layout description that is exchanged between the rendering client program in the client and the server.

Interface architecture

The interface between the client and the server is an XML based protocol via http(s).

In both directions only changes of data are exchanged. This is especially important for

sending layout definitions from the server to the client. If there is no change in the layout then there is only a minimum of data that is sent from the server to the client.

Development process

The typical development process for creating a screen or a part of a screen is:

Your Name

Result

You create the layout as XML definition by using a WYSIWYG screen designer.

```
<t:rowdemobodypane rowdistance="5" >
  <t:row>
    <t:label
      text="Your Name" width="120" />
    <t:field id="g_ccpreview_4"
      text="#{d.Helloworld.name}"
      width="200" />
  </t:row>
  <t:row>
    <t:coldistance width="120" />
    <t:button
      actionListener="#{d.Helloworld.onHello}"
      text="Hello!" />
  </t:row>
  <t:rowdistance height="50" />
  <t:row>
    <t:label text="Result"
      width="120" />
    <t:field enabled="false"
      text="#{d.Helloworld.output}"
      width="100%" />
  </t:row>
</t:rowdemobodypane>
```

You create the code - supported by a code generator:

```
public class Helloworld
{
    String m_name;
    String m_output;

    public void setName(String value)
    { m_name = value; }
    public String getName()
    { return m_name; }

    public String getOutput()
    { return m_output; }

    public void onHello(ActionEvent ae)
    {
        if (m_name == null)
```

```
        m_output = "No name set.";
    else
        m_output = "Hello world,"
            +m_name+"!";
    }
}
```

That's it! - The result can either be directly started in the browser or can be re-used as screen component in other screens.

Tools

CaptainCasa Enterprise Client includes tools in the following areas:

- WYSIWYG screen designer
- Style manager
- Internationalization manager
- Code viewer, code generator
- Profiling tools

All Java application development is done in a Java IDE of choice, for example:

- Eclipse
- IDEA
- NetBeans

All artifacts - including the screen definitions, styles, ... - are managed as files within a project within the corresponding IDE. The IDE is used for synchronizing with source code repositories (SVN, GIT, ...).

Technologies used

Client side:

- JavaScript
- (own) Control Library
- http(s) communication via "AJAX calls"

Server side:

- Java
- Java-EE server (starting with Tomcat, Jetty, ...)

CaptainCasa in a nutshell

CaptainCasa GmbH

CaptainCasa GmbH was founded in 2007.

CaptainCasa Enterprise Client was rolled out since 2007, using a Java-Swing-Client as front-end.

In 2013 a JavaFX version of the client was published, expecting JavaFX to become the new Java front-end standard...

After several approaches to cover the rendering-complexity of typical business application clients with existing web frameworks the so-called RISC-method was invented in 2017. This method was a breakthrough for our client activities and first time enabled us to provide an adequate level of flexibility, performance and robustness within the browser.

Throughout all the different client technologies, CaptainCasa kept a strict upwards-compatibility. Still screens created in the year 2007 using a Java-Swing-client are 100% compatible to run in the web-client of the current version.

CaptainCasa GmbH is located in Bammental, close to Heidelberg, Germany.

CaptainCasa Community

The users of CaptainCasa Enterprise Client form an active community which is tied together by a forum and by an annual physical

meeting in Heidelberg. In 2019 the 14th Community Meeting was held.

Licensing

CaptainCasa Enterprise Client can be used for free - without functional limits or usage restrictions.

Commercial licenses are available and add:

- Source code access
- Investment security
- Service & support

Education

Just Java is enough to start developing with CaptainCasa Enterprise Client.

CaptainCasa provides training-workshops with a typical duration of 2 days. After the workshop you will have a deep understanding of CaptainCasa Enterprise Client and you will have some concrete idea how to integrate the interaction-part of your application with the logic-part.

Services

CaptainCasa provides project development services by tightly bound partners.

This includes attractive, approved possibilities to outsource development activities to Bulgaria: there is an established infrastructure available to quickly and efficiently start implementation projects.

CaptainCasa GmbH
Hindemithweg 13
D-69245 Bammental
<http://www.CaptainCasa.com>
info@CaptainCasa.com