Developer view of DomUI

A quick summation of facts about DomUI

Implementation language, supported browser, tools used

- DomUI is mostly written in Java, with small parts of Typescript, Javascript and scss (sass).
- It supports all standard modern browsers and Internet Explorer from version 10.
- It uses generics as much as possible, and uses annotations where useful too
- It uses a small Typescript part which is generic, and has some extra Javascript/typescript code to support complex components.
- DomUI uses JQuery as the backing Javascript library for many functions.
- A set of base libraries containing common Java functionality is part of DomUI, so that DomUI does not have a huge amount of dependencies

Core concepts

Screen manipulation: DomUI's DOM

- DomUI screens build a browser-like DOM on the server, and manipulate that DOM to form the UI.
- The DOM is sent as normal HTML to the browser at initial page load.
- But any change after that is sent as an *optimal delta*, a set of commands that send the changes made server-side to the equivalent DOM inside the browser client.
- Since only delta's are sent DomUI is fast.
- And since DOM manipulation is done on the server, in Java, it is easy, type safe and can directly communicate with whatever Java code is present.
- This means: no need to write badly typed and error prone web services for the UI!

The last point is a large advantage over the current crop of Javascript based UI's. Using Java means that a lot of errors are caught at compile time and not at runtime. And since there is no web service layer that hides crucial information (like field types and even the presence and names of fields) this means that changes to code leads to compile time errors instead of runtime odd behavior.

DomUI - Java "interface"

- DomUI components and code prefer configuration over extension. This means that instead of creating a new class extending a DomUI class you just call setters on a DomUI class to configure it. Being able to configure programmatically is crucial to have good support for metadata-driven code: you cannot programmatically extend a class but you can easily configure from code.
- We use *configuration by exception* as much as possible. All components behave as reasonable as possible and are fully styled.
- DomUI tries to make the most of the (still pretty limited) Java type system:
    - All components that handle input are strictly typed, so that the Java code knows what type to expect from the component.
    - Most of DomUI's code is annotated with @NonNull and @Nullable annotations, and is set up so that the excellent Eclipse batch compiler can be used to do compile-time checking of null constraints. DomUI's Maven build uses the Eclipse compiler so that DomUI's code is compile-time checked for null errors for as much as possible.
    - Since Java's architects fail to solve the properties problem DomUI has a concept of "typed properties". These are classes automatically generated by an apt compiler extension which contain all the properties and their type of data classes. Using typed properties makes the code again way more safe: any refactoring of property names will cause *compile time* errors instead of runtime problems. In addition, because the properties are typeful, a lot of casts are no longer necessary.

Metadata

- DomUI has a metadata layer that uses all available information to decide how to show and handle data. The layer uses existing info (JPA /Hibernate annotations etc) and can be extended.

DomUI is an AJAX UI.

- DomUI uses optimal delta's to send AJAX updates as efficiently as possible. The delta calculation is very fast as we do not need to calculate a delta from the DOM directly; DomUI saves information on changes made in such a way that delta's can be generated very quickly.
- There is no XML "programming" involved in DomUI: most is just Java code. One usually only needs to code (some) Javascript when you integrate some Javascript component, like the Ace editor.
- It's very easy to create components or page fragments in DomUI. This makes it easy to build reusable and maintainable code.

Building an application

- A full DomUI application consists of a web.xml containing a DomUI <filter> specification plus a set of Java classes as a webapp.
- The Application class initializes the application, and defines the "root" DomUI page. It is an application level Singleton.
- Pages are Java classes that build a DOM that closely resembles the resulting DOM on the browser (layer 0).
- But DomUI has a lot of prebuilt components that itself are built using the DOM nodes (layer 1). Most DomUI code uses these components, not bare-bones HTML.

State management

- DomUI pages are stateful.
- This means: no serialization crap, no detached objects, no reload of data with every request.
- The database layer is closely integrated in DomUI and generalized so that backend technologies like Hibernate, JPA or just plain JDBC are all possible using the same data interface.
- DomUI pages have a "subsession" per "browser tab" and can be made completely cookieless: each page has a $cid parameter which contains a session ID which keeps track of the user's session. This also means that tabs/browser windows have (largely) independent sessions.

Integration within an application, "Hybrids"

- Creating a "hybrid" application, mixing DomUI and other display technologies, is not hard
  - It does require knowledge of DomUI's state management: the integration needs to make sure that DomUI pages are deleted from memory as soon as possible.
- The very large ERP application that DomUI was written for is a hybrid, combining JSP pages with AJAX pages and DomUI pages.

Accessing data

- DomUI integrates out of the box with Hibernate 5.2
- DomUI has an abstract and generic query interface, loosely modeled like Hibernate's Criteria, but without it's design flaws. By using this framework kludges like manually creating SQL are not needed in components that help with defining queries.
- This framework can be used for Hibernate (both JPA and native)- but also for just JDBC queries or queries against a collection-of-objects. Other frameworks can be added relatively easily.
- The framework is Java-centric, and uses the typeful properties to create type-safe queries.
- DomUI by default retains all data loaded for a page in a page-persistent session. This means that some care is needed for forms loading lots of data. But it also means that manipulating state is peanuts.
- It also means that forms have no problems with multiple roundtrips to the server: on return everything is where it was, without serialization horrors or detached status.

Which makes edit forms really simple: no models, no reloading, no serialization.

The DOM in DomUI

- Pages are created as a HTML DOM (Document Object Model) represented as Java objects. These objects form a tree in server memory, and by manipulating and rendering that tree we can render an HTML UI on the browser
- Actions executed by the user, like pressing a button, are sent to the server. The action is handled by Java code (in this case an IClicked<T> handler) which effects the action. This action then modifies the server-side DOM tree (because the Java code for instance adds a new row, or adds an error message, or whatnot).
- Changes made to the DOM are sent to the browser as a delta. This delta is generated conceptually by sending only the changes made to the tree. The OptimalDeltaBuilder is fast, and produces delta's that are as small as possible. For instance, if we delete 1000 rows of a 1001 row table it does not send 1000 "delete" commands but it sends a "delete" for the parent, then an "add" for the remaining row.
- Since the delta is small, updating the page is fast: it uses hardly any bandwidth.
- Calculating the delta is also very fast as the delta is assisted by collected change information.
- The UI is then created by creating a DOM that is a 1-on-1 match with the HTML that shows that UI. This means that HTML knowledge is required from a developer.
- One way to reduce the amount of "DOM code" to write is to use components. A component is a basic HTML entity (like a DIV or SPAN) that encapsulates complex structure and behavior.
- Because components themselves are simply DOM nodes, they are very easy to write, using only Java. This is important because writing complex pages as a set of components make them easy to maintain and build!
- Another way to reduce code is to use one of the provided builders, or to build your own. A builder is a class that helps you to create a certain UI, but which is not part of that UI. The most often used builder is the FormBuilder, which creates input forms.

DomUI Layering: layer 0

- DomUI is layered. Layer 0 is the core HTML layer.
- This layer defines most of the core HTML tags, like Div, Table, TD etc
- Each code tag node is either a NodeBase (cannot have children) or a NodeContainer (can have children).
- These base nodes encapsulate all common behavior, and handle all CSS styling. CSS properties are exposed as java class properties, and can be changed at will at runtime to provide a component's *dynamic* behavior.
- Layer 0 nodes do not have "extra" functionality; they are as close to the original HTML elements as possible.
- Layer 0 elements are the "building blocks" for everything; they are used in components and usually to create the large scale structure of a form.

Layer 1

- Layer 1 is the component layer.
- This layer consists of components that are mostly constructed out of layer 0 nodes.
- A component exposes behavior and can be an input component.
- All DomUI input components are strongly typed, and always return their value in the appropriate Java type. For instance the Text<T> control is an html <input type="text"> component but can be used as Text<Integer>. In this case the getValue() call will return that Integer always (or null if the control is empty).
- This makes conversion and (field level) validation a function of a control itself. The control uses the validator framework and converter framework to do the actual work. Most common conversions are predefined.