Waltz Architecture

Table of Contents

Overview

Waltz is a web based application for viewing and documenting and defining the technology landscape within large organisations. This document is *not* a function overview of the application. If you are interested in a general Waltz overview please read through the Introduction to FINOS Waltz article.

This document will take a high level view of the Waltz application. We will focus on the architecture of the client side app, the server and the database. By the end of the article you should have a good understanding of how the various modules interact and where to have a better idea of how changes can be made to the codebase.

Intended Audience

The primary audience for this document are:

- Developers, modifying the Waltz codebase itself
- Integrators, importing / exporting data sets into Waltz
- Administrators, to gain a better understanding of how the internals of the application function

Modules

As can be seen in the diagram below Waltz follows a traditional *n-tier* architecture. The user interacts with the Waltz application via a browser based single page application. Requests and responses between the client and server follow a RESTful pattern using JSON as the serialization mechanism. The server side endpoints route the request to the appropriate object in the service layer which, in turn, will orchestrate calls to the data access layer.

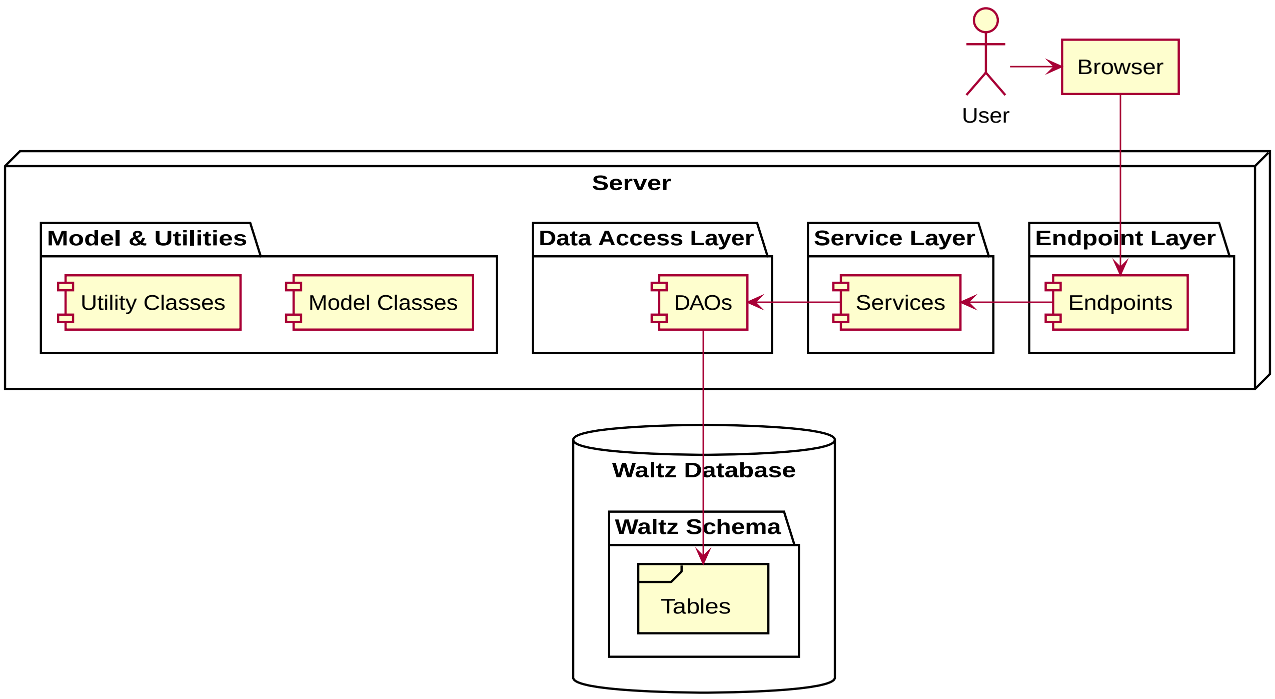

Figure 1. Waltz high level architecture

We will now take a closer look at each of the high level modules, starting with the browser and working our way to the database. Each section can be read in isolation or in sequence, we shall try to minimize forward references when discussing topics.

By necessity, we can only provide so much detail, please reach out to us via our Github issues page with any specific questions.

Browser

The Waltz user interface has been developed as a single page application using the AngularJS framework. We are currently investigating migration options as we would like to move away from AngularJS since it is now in maintenance-only mode .[1] In addition to AngularJS the application extensively uses the Lodash [2] and D3 [3] libraries for processing data and visualizations. The presentation layer is based upon Bootstrap [4] with custom styling provided via Sass [5] stylesheets.

Browser Code Overview

The `waltz-ng` project stores all of the browser code. It is split into several folders:

Web application directory structure

```
waltz-ng/
    |- client/      -- large majority of js code and templates
    |- images/      -- static images, not many of these, mostly logos.
    |- style/       -- global sass stylesheets
    |- test/        -- mocha unit tests
    | index.ejs     -- main app entry point
    | package.json  -- js dependencies
    | webpack.*.js  -- build scripts
    | ...           -- assorted other configuration files
```

The `client` folder contains almost the code for the browser application and consists of several hundred components. The client code is split across many sub folders each of which represent an angular module. Each of these sub folders has a predictable naming convention:

Angular module structure

```
waltz-ng/client/
  |- some-module/              -- module name, typically domain/function aligned
      |- components/
        |- comp1/              -- each component has it's own directory
          |- my-comp1.js       -- component files names should match their identifiers
          |- my-comp1.html     -- template file should shares the same name
          |- my-comp1.scss     -- (optional) stylesheet should shares name
        |- ...
      |- pages/                -- (optional) top level view pages
        |- pagename/           -- each view page has it's own directory
          |- pagename.js       -- split into the core view logic and..
          |- pagename.html     -- the view template
        |- ...
      |- services/             -- (optional) services & stores
        |- some-store.js       -- stores interact with the server
          |- ...
      |- index.js              -- registers components and stores (indirectly, routes)
      |- routes.js             -- (optional) registers /pages this module contributes
```

The `waltz-ng/client/modules.js` file registers each module and ensures they are loaded when the client initialises. Stylesheets are registered in the `waltz-ng/styles/_waltz.scss` file using the Sass import mechanism.

NOTE The `playpen` module is reserved for developer testing. It gives a space for quicly prototyping new features without having to create lots of bolierplate code for registering modules, components and services.

Basic component structure

Each component js file typically follows a simple structure. An example component file is shown below.

Component structure

```
import { CORE_API } from "../../../common/services/core-api-utils";
import { initialiseData } from "../../../common";

import template from "./my-comp1.html";

const bindings = {
    parentEntityRef: "<"
};

const initialState = {
    list: []
};

function controller(serviceBroker) {
    const vm = initialiseData(this, initialState);

    const loadStuff = () => {
        return serviceBroker
            .loadViewData(
                CORE_API.SomeStore.loadSomething,
                [vm.parentEntityRef])
            .then(r => vm.list = r.data);
    };

    vm.$onChanges = () => {
        loadStuff();
    };
}

controller.$inject = [
    "ServiceBroker"
];

const component = {
    template,
    bindings,
    controller
};

export default {
    component,
    id: "waltzMyComp1"
};
```

The component file exposes the component and it's identifier (line 40). The component in turn consists of a bindings (passed in params), controller and template (line 34). Angular provides dependency injection services, so we wire any required services to the controller 'constructor' (lines 30 and 14).

NOTE We do not rely on name introspection to provide the injected services as the minification build step breaks this mechanism. We explicitly switch this off in `waltz-ng/index.ejs` via usage of the `ng-strict-di` directive.

By convention, we use the `vm` variable to refer to the component instance and initialise it with any sensible defaults (line 15). Since the attributes of this vm variable are not be formally declared (e.g. with `let` or `const`) we strongly encourage variables to be defined in the `initialState` object (line 10).

WARNING A few, seldomly used, old components in Waltz use the older, directive based approach. This is more verbose and cumbersome than the newer component based approach. The main advantage of the older approach is being able to declare components at the attribute level instead of at the element level. This is used in the drop down navigation bar where html element structures cannot contain intermediary , angular introduced, elements.

Component Naming conventions

In angular components are registered with an identifier. All waltz component identifiers are prefixed with `waltz`. This ensures there will be no namespace collisions in the html files with other components registered by third party libraries.

In html templates angular uses *kebab-case* for elements (e.g. `waltz-my-comp1`). The part after the `waltz-` prefix *should* be an exact match of the filenames corresponding to the component, template and stylesheet. Similarly, when registering components, angular uses *camelCase*. In this example the identifier would be `waltzMyComp1` (see line 42 in the component listing).

Following this convention allows developers to quickly navigate the codebase using simple file searches. When the convention is broken, developers need to grep the contents of files. This is time consuming and error prone.

Server

Database

[1] See AngularJS Version Support Status

[2] A collection of utilities to help process data in javascript applications, see the Lodash website

[3] A data visualization framework for creating interactive graphics, see the D3 website

[4] Currently using Bootstrap 3.x

[5] A language which compiles to CSS, see Sass website for more information