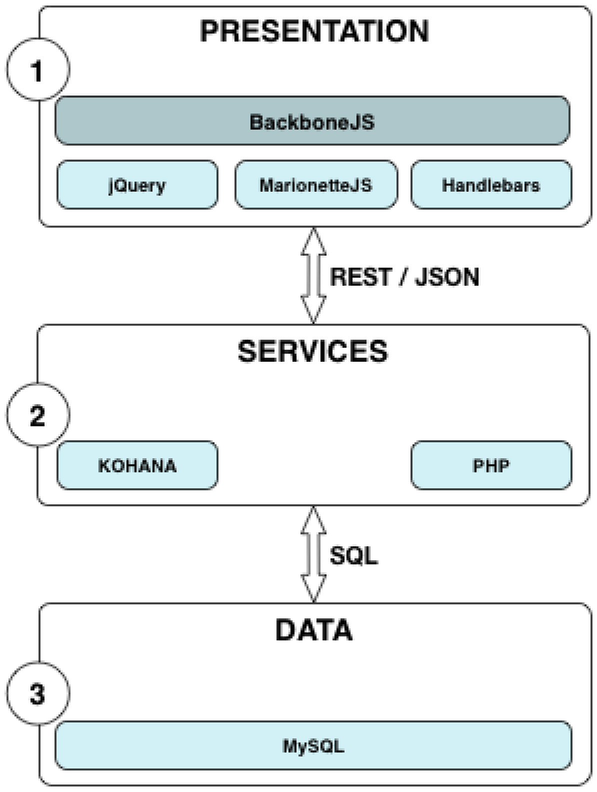
Code Organisation

The Ushahidi Platform is split into 3 layers: Presentation (Frontend), Services (API), and Data. The API layer consists of a core application and a delivery mechanism. The core application is pure object-oriented PHP and the delivery mechanism is a PHP application built using the Kohana Framework. The Frontend is a javascript application, built on BackboneJS.



In theory the Kohana application could handle frontend views and interactions too, but splitting the API out allows us far greater flexibility in the variety of applications that we can build. Mobile apps, 3rd party integrations, etc are not 2nd class citizens: they can consume the same API as our primary frontend. The API has to be able to do everything to our data.

Containing the core business logic within a core application that is separate from the Kohana delivery layer allows us to test the core application, independent of the database (or any other implementation details) while also enforcing the internal API by which the rest of the system operates. This allows us to modify internal details, such as the database structure, without breaking the external API, as well as ensuring that the entire system remains in a stable, tested state.

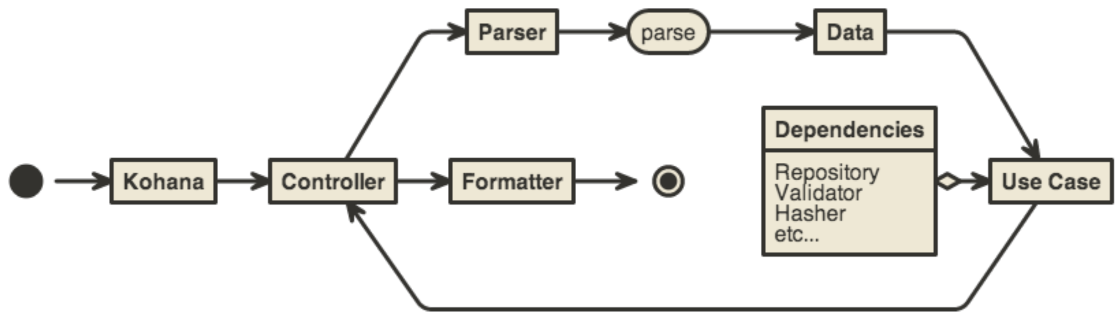
- API Delivery
- Core Application
    - Models
    - Controllers
        - Base API Controller
- Frontend
    - Startup execution
        - config/Init.js
        - App.js
        - Router and Controller
        - Layouts and Regions
            - AppLayout
    - Getting the Frontend UI to the browser
        - Main controller
        - Kohana Media module
    - OAuth
- File structure: what lives where?

API Delivery

Within the API there are two layers: the delivery and the business logic (core application). The delivery layer follows a Model View Controller (MVC) pattern, with the View consisting of JSON output. The Controllers use a Service Locator to load and execute various tools, taking the API request inputs and returning the requested resources.

Core Application

Within the core application, we use generally follow the Clean Architecture. The central part of the business logic is defined as use cases and entities. All dependencies flow inwards towards the entities, which have no dependencies. In order to bring user input to the use cases, we create a Data Transfer Object (DTO) and pass this object from the delivery layer into the use case. The DTO is a very simple object which is defined by the use case and contains all of the possible inputs for that specific use case. In order to create the DTO, we use a Parser object. The DTO is then checked for consistency by using a Validator. Once validated, the use case will complete its execution and return the resulting entities back to the delivery layer for conversion via a Formatter. (For a longer overview of clean architecture and how it is implemented, please read Ushahidi Platform: Under the Hood, Part 1). Data flow within the platform can be visualized as:



Note that this diagram is not ideal and will be changing before the final version of the platform. Specifically, data is flowing backwards to the controller and should be flowing directly into the formatter. We will be addressing this in the near future.

Models

Modeling within Ushahidi is primarily handled at the core application level with entities. Each entity also defines a read-only repository interface for the database access layer. The writable repository interfaces are defined by each use case that requires storage. The readable and writable interfaces are implemented within the delivery layer, to keep the core application free of storage details. Typically, the implementation of multiple related interfaces will be done by a single object. For instance, the user storage object will implement the `UserRepository` to read users, as well as the `UserRegisterRepository`, `UserLoginRepository`, `UserUpdateRepository`, etc. Repository read operations will always return an entity, or collection of entities. Writing operations can have various output, depending on the situation.

Verification of incoming data for writing operations is first parsed into a DTO, which is then validated by the use case.

Formatting of entities into consumable resources is done at the delivery layer.

Controllers

Each request to the API is routed to an API controller, and action. The controller code processes any user input and returns generates a JSON response. The controller actions are mapped based on the type of HTTP request (GET, POST, PUT, DELETE, etc):

HTTP	Description	Action
`GET /foo`	get collection of "foo" entities	action_get_index_collection
`GET /foo/:id`	get "foo" entity of "id"	action_get_index
`POST /foo`	create a new "foo" entity	action_post_index_collection
`PUT /foo/:id`	update "foo" entity of "id"	action_put_index
`DELETE /foo/:id`	delete "foo" entity of "id"	action_delete_index

Every controller that responds to API requests will follow this pattern.

Base API Controller

All API controllers are built off a base controller - `application/classes/Ushahidi/Api.php` - which handles a large amount of the repeated code in the API. The base controller takes care of:

1. Initial access checks (oauth2 scope, general access to this resource type)
2. Parsing the incoming request. We handle GET/POST/PUT/DELETE requests through different controller actions, and split collection from individual resource handlers. For example:
   - POST /api/v2/posts is routed to Controller_Api_Posts::action_post_index_collection();
   - GET /api/v2/tags/1 is routed to Controller_Api_Tags::action_get_index();

   The second part of this is parsing the request data. For GET requests theres not much to do here, except for check any query params for sorting/filtering/etc. For PUT/POST requests the base controller handles decoding the JSON passed in the request body.
3. Formatting the outgoing response. Individual API controllers only have to set $this->_response_payload to a PHP array. The encoding of this data is left up to the base controller. The base controller will check the format parameter and encode the data as either JSON or JSONP, as requested by the query string `format` parameter.

Frontend

The frontend for Ushahidi 3.x is a javascript application (in the browser) built on BackboneJS (& Marionette JS). All data for the application is loaded from the API. The javascript itself is broken down into many 'modules' which are loaded by RequireJS.

The javascript application lives in modules/UshahidiUI/media/js. The structure here is:

lib/ - 3rd party libraries (jquery, backbone, etc)
app/ - All custom application modules

collections/ - All collection classes
controllers/ - All controllers, currently only a single controller in Controller.js
routers/ - All routes, currently only a single router in AppRouter.js
views/ - All view classes
templates/ - Handlebarsjs templates used by the views
util/ - small utility modules
App.js
config/Init.js

tests/ - Frontend test (not yet built out)

Startup execution

config/Init.js

Init.js is first file loaded by requireJS. It sets up some initial RequireJS config (paths, and shims for non AMD modules), requires other App files and starts the App. It's simple enough to include most of its code below

```
// Cut down excerpt from Init.js

require.config(
{
 baseUrl : "./media/kohana/js/app",
 paths :
 {
  // Set paths to libraries
 },
 shim :
 {
  // Shim non AMD modules
 }
}

require(["App", "routers/AppRouter", "controllers/Controller", "jquery", "jqueryui", "backbone.validateAll"],
 function(App, AppRouter, Controller) {
  App.appRouter = new AppRouter(
  {
   controller : new Controller()
  });
  App.start();
  window.App = App;
 });
```

App.js

This is the module for the main application object.. there's not actually a lot that goes on here: we create the object, add regions, save our config and oauth objects for later, and add an 'Initializer' callback. The initializer callback is fired when the application starts, and all it does is start Backbone.history..

Router and Controller

Once the App class has started, most control is handed over to the router and controller class. The router maps a url to a controller and action, at the moment these mappings are all fairly simple.

```
// AppRouter.js
define(['marionette', 'controllers/Controller'],
 function(Marionette, Controller) {
  return Marionette.AppRouter.extend(
  {
   appRoutes :
   {
    "" : "index",
    "views/list" : "viewsList",
    "views/map" : "viewsMap",
    "posts/:id" : "postDetail",
    "*path" : "index"
   }
  });
 });
```

Each route maps directly to a function in the controller. The controller handles switching layouts and regions, creating views and binding models or collections to these.

I've shown part of the Controller.js file below. The 'initialize' function is run when the App first starts. It creates an 'AppLayout' - this is kind of a special view with several regions

```
// Controller.js
  Backbone.Marionette.Controller.extend(
  {
   initialize : function(options) {
    this.layout = new AppLayout();
    App.body.show(this.layout);

    var header = new HeaderView();
    header.on('workspace:toggle', function () {
     App.body.$el.toggleClass('active-workspace')
    });

    this.layout.headerRegion.show(header);
    this.layout.footerRegion.show(new FooterView());
    this.layout.workspacePanel.show(new WorkspacePanelView());

    App.Collections = {};
    App.Collections.Posts = new PostCollection();
    App.Collections.Posts.fetch();
    App.Collections.Tags = new TagCollection();
    App.Collections.Tags.fetch();
    App.Collections.Forms = new FormCollection();
    App.Collections.Forms.fetch();

    App.homeLayout = new HomeLayout();
   },
   //gets mapped to in AppRouter's appRoutes
   index : function() {
    App.vent.trigger("page:change", "index");
    this.layout.mainRegion.show(App.homeLayout);

    App.homeLayout.contentRegion.show(new PostListView({
     collection: App.Collections.Posts
    }));
    App.homeLayout.mapRegion.show(new MapView());
    App.homeLayout.searchRegion.show(new SearchBarView());
   },
   // etc
  });
```

Layouts and Regions

All the HTML in the Ushahidi 3.x UI is built using backbone views and handlebars templates. We use MarionetteJS Layouts and Regions to combined many nested views. The base views and regions are managed through AppLayout and and HomeLayout. A region is basically a wrapper around a particular DOM element to make it easy to show/hide a view inside that DOM element. A layout is basically a view with several regions bound to it, ie. we use a view to render some HTML then bind several regions to parts of the view's HTML output.

AppLayout

The AppLayout is the top level view for the application. It populated the <body> tag with HTML and create the header, main, footer, workspace and modal regions. The header, footer and workspace regions mostly keep the same views. The modal region is used for modal popups like create or edit post. There are a few different views/layouts we swap in and out of the main region: HomeLayout, PostDetailLayout, SetListView, etc.

Getting the Frontend UI to the browser

One thing you will notice if you dig into the code, is that the docroot (httpdocs/) actually only contains a small number of files, and all requests are passed to index.php. The frontend (JS/CSS/HTML/etc) is still served up by the same Kohana application as the API. The frontend actually lives in a Kohana module: UshahidiUI. This handles a couple of things:

- the base route '/', controller and views
- serves 'media' files: javascript, css, images, fonts.

Main controller

This controller handles requests for '/' - the main page of a deployment. This controller does very little real work: its builds an array of config data, and renders a single view - 'modules/UshahidiUI/views/index.php'.

```
abstract class Ushahidi_Controller_Main extends Controller_Template {

    public $template = 'index';

    public function action_index()
    {
        $this->template->site = array();
        $this->template->site['baseurl'] = Kohana::$base_url;
        $this->template->site['imagedir'] = Media::uri('/images/');
        $this->template->site['cssdir'] = Media::uri('/css/');
        $this->template->site['jsdir'] = Media::uri('/js/');
        $this->template->site['oauth'] = Kohana::$config->load('ushahidiui.oauth');

    }

}
```

This view generates the starting HTML for our application, including tags to load JS/CSS files and turning that configuration array in a json object.

After the browser loads this view, everything is hanled by the javascript application, and the API.

Kohana Media module

OAuth

OAuth - handled as part of the API, mostly abstracted to a module, etc

File structure: what lives where?

- src
    - Ushahidi
        - Entity - Application entities
        - Tool - Basic tools and tool interfaces
        - Traits - Reusable traits
        - Usecase - Application use cases, Data objects, and Repository interfaces
- spec - PHPspec tests for core application
- application
    - classes

    - Ushahidi
      - Api.php - Abstract base class for all API controllers
    - Controller
      - Api - All API controllers
  - config
    - environments - environment specific config overrides
    - auth.php - config for kohana auth module
    - database.php - database config
    - init.php - init config, passed to kohana::init()
    - media.php - config for kohana-media module
    - modules.php - module paths to be loaded
  - migrations - database migrations
  - routes
    - default.php - Default route definitions, can be overridden per environment
  - tests
    - classes - PHPUnit tests
    - features - Behat tests and contexts
    - datasets - Datasets used in testing
  - views
    - api - api error views
    - oauth - views for oauth authorization flow
  - vendor - vendor libraries ie. gisconverter
  - bootstrap.php - Kohana bootstrap file: loads and configures the kohana framework.
- modules
  - UshahidiUI
    - classes
      - Controller
        - Main.php - Main controller: handles frontend routes (/<action>)
      - Ushahidi
        - Controller
          - Main.php - Base class for main controller
    - config
      - ushahidiui.php - UI config, passed as JSON object to frontend JS
    - media
      - js
        - lib - JS libraries
        - app - Application JS. All files are loaded as requireJS modules
          - collections - BackboneJS collections
          - config
            - Init.js - Require JS Init files, configures requirejs and initializes the app.
          - controllers - controllers
            - Controller.js - application controller, currently the only controller
          - models - BackboneJS modles
          - routers - Backbone JS routers
            - AppRouter.js - application router, currently the only router.
          - templates - handlebarsjs templates
          - util - small utility modules
          - views - BackboneJS views
          - App.js - Main Application module: sets up main application class with regions, initializers etc
        - tests - Frontend tests (not used yet)
      - css - Frontend CSS, compiled and 3rd party css files (such as fontawesome)
      - images - images for the frontend ui
      - font - font files for the frontend ui
      - scss - Source SCSS files
    - views
      - index.php - frontend view: contains just the initial html, with an empty body.
    - init.php - UshahidiUI module init script, add a single frontend route.
    - Gruntfile.js - Grunt config, used to build SCSS files to CSS, compile requirejs files, etc
- httpdocs
  - index.php - Loads and bootstraps Kohana..
  - template.htaccess - .htaccess template file, copy to .htaccess and customize

- vendor - Composer modules