University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

BACHELOR THESIS

Dynamic Webform Builder

Scientific Adviser:
ing. Andrei Picuș
as. drd. ing. Laura Mihaela Vasilescu

Author:
Flavius-Costin Tîrnăcop

Bucharest, 2015

Thank someone?

Dedicated to someone?

Abstract

Development time is an important topic today, with companies having very strict roadmaps. By separating the dependencies between the front-end and back-end and by generating UI components in an automated way, developers can gain drastically speed up the development process.

The project aims to provide a front-end application that is agnostic of the back-end structure, providing independence between the two platforms. The application will also be able to auto-generate generic UI components regardless of the data fetched from the API.

Keywords: Web, Front-end, HATEOAS, REST, API, React, Django, Python

Contents

List of Figures

Chapter 1

Introduction

1.1 Objectives

The Dynamic Webform Builder project started out as an idea at Hootsuite Romania and had the main goals of:

- Defining and maintaining a REST (Representational State Transfer) API (Application programming interface) and model schema in a single place (server-side)
- Auto-exploring the resource and auto-generating generic forms
- Creating add/edit forms with minimal client-side work (scalable user interface)
- Have consistent look and experience in all forms

1.2 Motivation

Today's web applications are split into the front-end, usually available on the client side and the back-end often described as the server-side. This division forces developers in big companies to work in specialized teams for the front -end and the back-end development. Ideally developers wish that the front-end and the back-end of the application are agnostic of each other, therefore excluding further problems such as changing a resource URL(Uniform Resource Locator) on the server side and having to manually change it everywhere in the front-end. Avoiding these kind of conflicts will speed up the development process and therefore generating more profit for the company.

For a front-end application to be agnostic of it's back-end it should be able to auto explore a given API endpoint that may change over time. Also the front-end should be able to auto-generate UI components regardless of the data type and structure retrieved from the server.

Therefore this project aims to demonstrate that it is possible to have auto-exploration and auto-generation in a front-end application that is independent of the data changes from the back-end server.

1.3 Background

In this section we will describe the base concepts that are used across this thesis. Therefore we will start by describing the concept of a Web service in section 1.3.1 followed by explaining the base concepts behind the REST software architecture in 1.3.2 and it's HATEOAS constraint described in 1.3.3.

1.3.1 Web service

As the W3C (World Wide Web Consortium) states [2], "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards." Therefore we can define a web service in more simpler terms as a framework for a conversation between two computers that are communicating over the web, where a client sends a request message server receives that request, processes it and returns a response message.

Interaction with web services is done typically trough HTTP (Hypertext Transfer Protocol). An example of web service interaction in today's web pages is described by AJAX(Asynchronous JavaScript and XML) requests. To prevent a full page transition a the browser may initiate an asynchronous request (AJAX call) asking for specific data to be rendered later(eg. rendering an email into the Gmail application). The server will send a response in the form of HTML (HyperText Markup Language) code or just pure data encoded in XML (EXtensible Markup Language) or JSON (JavaScript Object Notation).

Web API

Even if the messages sent and received from the web service are transmitted trough HTTP, as a developer you still need to have knowledge of the messages format that are being sent or received. Therefore you will need to know details about the service's API.

An Application Programming Interface (API) is a particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API.

A Web API defines both the server-side API as well as the client browser API and must impose specific elements like the message format or request syntax. Some web services may use SOAP, XML or JSON for data formatting on the request and response calls. On the client side a developer should know if the request syntax may imply using certain URI's (Uniform Resource Identifiers) and specific parameters and data types. On the server side the web service will act on different types of requests that can be associated with HTTP verbs like GET, POST, PUT, PATCH or DELETE. Again, the developer should know the format of the messages received and structure of the data including specific fields and data types.

According to the W3C [2] we can identify two major classes:

- REST-compliant Web services, in which the primary purpose of the service is to manipulate representations of Web resources using a uniform set of stateless operations.
- Arbitrary Web services, in which the service may expose an arbitrary set of operations.

Today web API's have been moving[1] to a much simpler and well defined representational state architecture and are now called RESTful API's. These systems usually communicate over

[1] http://www.infoq.com/articles/rest-soap

HTTP with the same HTTP verbs (GET, POST, PUT, PATCH, DELETE) used by browsers to render web pages.

1.3.2 REST

TODO:
Hardcore theory

1.3.3 HATEOAS

TODO:
Hardcore theory

Chapter 2

State of the art

In this chapter we will start to describe the technologies and frameworks used for both back-end and front-end implementations, followed by presenting how they interact as a system in the next chapter 3. For serving the web-page we used Django 2.1.1 and for delivering the and the REST API we used a Django compatible extension called Tastypie 2.1.2, the Django compatible solution. For the display logic and for rendering the front-end components we used Facebook's React.js framework 2.2.1 and also Bootstrap/Flexbox 2.2.2 for styling.

2.1 Back-end Frameworks

2.1.1 Django

For our back-end framework we used Django mainly because of it's performance, scalability, stability and the flexibility that it gives to developers. Being a free and open source project, Django benefits of a large community support therefore having an extensive official documentation, a vast collection of tutorials available online and many more extensions available to install trough the built in PIP (Python Package Index) installer.

Django is a web application framework written in Python with a "batteries-included" philosophy. The principle behind batteries-included is that the common functionality for building web applications should come with the framework instead of as separate libraries. Therefore Django can provide developers with all the tools needed to put up a web page. It has a built in relational database for storing information, a templating system for rendering pages and a URL dispatcher for resolving page requests. The framework also comes with more advanced features like XSS (Cross site scripting), CSRF (Cross site request forgery) and SQL Injection protection but also an authentication and admin panel modules.

Django's architecture is described by the developers[1] as a MTV, which stands for "Model-Template-View„ even if it closely resembles a classic MVC (Model View Controller) pattern which many may be familiar with. A high level view of Django's architecture can be seen in Figure 2.1.

[1] https://docs.djangoproject.com/en/1.8/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names

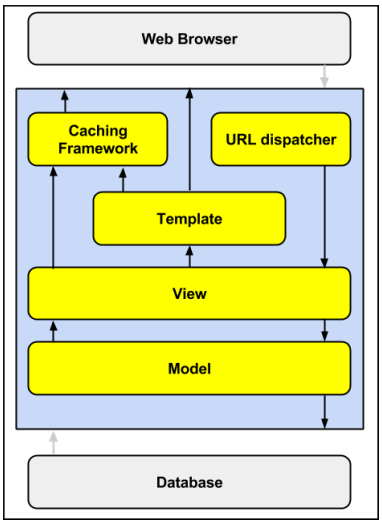

Figure 2.1: Django architecture

Model

The "Model" consists of an object-relational mapper that translates data models defined in Python classes to classic relational database tables. Django's default relational database management system is SQLite.

Models are represented by Python classes and contain essential fields and behaviors of the stored data and represent the single source of information about that data. Every class inherits the **django.db.models.Model** and each model attribute represents a database field. An very basic example can be seen in Listing 2.1.

```
1 from django.db import models
2
3 class Post(models.Model):
4     author_name = models.CharField(max_length=30)
5     text = models.CharField(max_length=200)
6     created_at = models.DateTimeField('created_at')
```

Listing 2.1: Post model class

We can see that we have two character type fields and one date type field that map to the corresponding columns in Listing 2.2.

```
1 CREATE TABLE post_table (
2     "id" serial NOT NULL PRIMARY KEY,
3     "author_name" varchar(30) NOT NULL,
4     "text" varchar(200) NOT NULL,
5     "created_at" timestamp with time zone NOT NULL
6 );
```

Listing 2.2: Post SQL table

Template

The "Template" consists of a web templating system that handles the page rendering on the client part. This is also why it is called the "Template" even if it basically resembles the MVC "View" component. A Django template is composed of the static parts of the desired HTML output as well as special syntax that describes describing how dynamic content will be inserted. An example of rendering a poll question with it's available choices can be seen in Listing 2.3.

```
<h1>{{ question.question_text }}</h1>
<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

Listing 2.3: Using Django templates

We can see that the Django that trough `{{var}}` specific variables are inserted into the HTML that will be generated, like the question text or the choices text. Also, Django provides specific built-in template tags like `{% for in %}` for looping over arrays, handling conditional statements, filtering etc.

View

The "View" is a regular-expression-based URL dispatcher that maps the URL to a view function and calls it. The view function can also check if a cached version of the page is available and skip the following steps. The component was named "View" by the developers because the callback function of the dispatcher describes which data is presented to de user. The "View" can be seen as an MVC "Controller" because the view functions performs the requested action which usually implies reading or writing to the database. An example of a urls.py file can be seen in Listing 2.4.

```
from django.conf.urls import include, url
from django.contrib import admin
from posts import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^posts/', include('posts.urls')),
    url(r'^admin/', include(admin.site.urls)),
]
```

Listing 2.4: Urls.py file

We can see that each URL pattern is matched via the regular expression set as the first parameter of the `url()` function (eg. `r'^$'`) and calls the function callback (the second parameter) with following optional parameters. In the first case the regular-expression will match the empty string with the index function from the views module.

2.1.2 Tastypie

Tastypie[1] is a webservice API framework for Django. It provides a convenient, yet powerful and highly customizable abstraction for creating REST-style interfaces. Like Django, Tastypie

[1] http://tastypieapi.org/

is an open source, community backed project. Some of it's main features are:

- Full GET/POST/PUT/DELETE/PATCH support
- Designed to be extended at every turn
- Includes a variety of serialization formats (JSON/XML/YAML/bplist)
- HATEOAS by default
- Well-tested and well-documented

Tastypie's basic functionality is taking data represented in the Django models (described by python classes), serializing it and sending the resulted data to the client that consumes the API. During the request/response cycle, Tastypie uses a large portion of the standard Django bahaviour and adds on top of that a „hydrate/dehydrate" cycle. An example for requesting data from a sample endpoint like `api/v1/author/?format=json` can be summarized as follows:

- The Django URL dispatcher checks the requested URL in the `Resource.urls` and on a match for the list view calls `Resource.dispatch`
- The `dispatch` method checks if the request is valid by establishing if the used HTTP method is valid, if the requesting user is authenticated and authorized and if the resource has a handle request method. If all checks are passed the `dispatch` calls `get_list`.
- The `get_list` method fetches the available ModelResource objects, sorts them and applies `full_dehydrate` on each one of them.
- The `full_dehydrate` has the purpose of taking the data from the ModelResource objects and transforming it into a much simpler Tastypie specific data structure, usually a dictionary of simple data types and. Therefore the method applies each object's `dehydrate` method to get a simpler data structure for serializing. The process is similar on POST/PUT/PATCH requests where the `hydrate` method is applied instead, hence the „hydrate/dehydrate" cycle.
- In the end, the `create_response` method serializes the data given to it in the requested format (XML, JSON) and returns HttpResponse (200 OK) with the resulted data.

2.2 Frontend-end Frameworks

2.2.1 React

"React (React.js) is a JavaScript library for creating user interfaces."[1] Built at Facebook and Instagram, and later open-sourced to the community (2013), React renders UI components and responds to user events, thus making it resemble the „View" component in the MVC architecture style. React aims to solve the problem of "building large applications with data that changes over time"[1].

The building blocks of React are its **reusable components**. What makes them so special is the ability to store state and data to always keep in sync with the UI. As Pete Hunt often says in his talks "Data changing over time is the root of all evil"[2], strengthening the idea that the main problem of UI design is knowing in what state it is at a moment in time. By encapsulating state in the components developers were able to reduce coupling and increase cohesion. An example of how a react component is structured can be seen in Listing 2.5.

[1] http://facebook.github.io/react/docs/why-react.html
[2] http://endot.org/notes/2014-01-30-react-rethinking-best-practices/

```
1  var InputComponent = React.createClass({
2    getInitialState: function() {
3      return {value: this.props.val};
4    },
5    handleChange: function(event) {
6      this.setState({value: event.target.value});
7    },
8    render: function(){
9      return (
10       <input type="text" value={this.state.value} onChange={this.
             handleChange}/>
11     );
12   }
13 });
14 React.render(<InputComponent val="Default value" />, document.
     getElementById('content'));
```

Listing 2.5: Simple react component

In the above example we can see a simple component that describes an HTML input field. Every React component is declared as a React class trough `React.createClass` function. The most important elements of a React component are:

- `props` - Props are usually received from the parent component are should never be modified inside the component's functions as they are the component single source of truth that it has.
- `state` - Describes the state of the component at a moment in time. The state can **only** be set at the initial rendering of the component trough `getInitialState()` function and trough the component `setState()` function. On any change of the state (resulted by calling the `setState()` function) the component will call the `render()` function, thus synchronizing the UI with its current state.
- `render` - The function that actually generates and updates the DOM components. It should never be called by the developer as it is automatically called on specific events such as component initial mount, `setState()` calls etc. Another feature that can be seen in the example is the return value of the `render()` function which actually contains some HTML syntax. This is an optional feature of React that js called JSX. The JSX syntax provides developers a more familiar way to describe the DOM elements that they deserve. In Listing 2.5 the return value will be compiled by a JSX compiler to the equivalent call: `React.createElement(type:"text", value:this.state.value, onChange:this.handleChange)`.

The only data a component will work with are `props` that are usually received from its parent component and the component `state`.

We talked about the React components, the main blocks of the framework but next we will explain how React is structured and how it uses this components to modify the DOM on user actions. In Figure 2.2 we can see a basic overview of the React architecture. The render cycle can be summarized in four steps:

1. When the user interacts with the browser it creates an event, for example a button is pressed thus sending a native browser event to the `ReactEvent` system.
2. Browser events may not be standardized between certain browsers so React takes this event, normalizes it to the W3C standards and creates a `Synthetic event` that is dispatched to the components that have a callback set on that event in the `User code`.

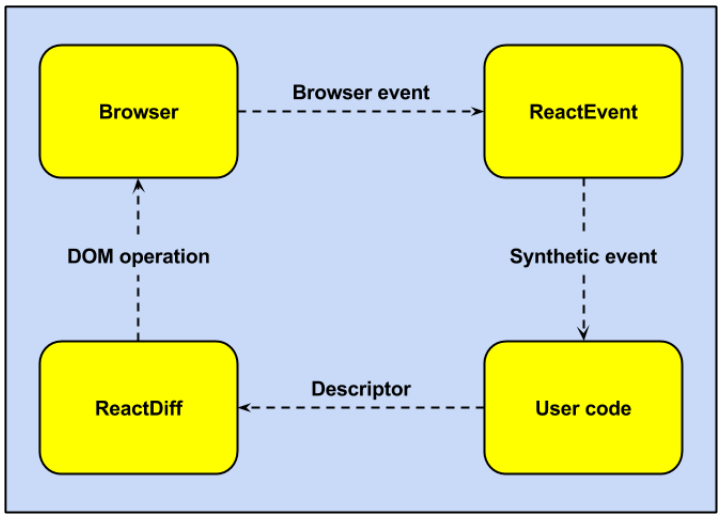

Figure 2.2: React architecture

3. Next the user code (that runs inside a react component) may change its state and call `setState()` thus triggering a re-render. At this step react creates a `Descriptor` with the element that has changed in the Virtual DOM (which we will explain next in more detail) and sends it to the `ReactDiff` algorithm.
4. Finally the `ReactDiff` algorithm computes the minimum number of DOM mutations and sends the operations back to the browser.

Virtual DOM

As mentioned before React holds a JavaScript in-memory representation of the web-page's DOM, called a **Virtual DOM**. As seen in Figure 2.3, React components will first mutate the Virtual DOM that will later be used for diffing. With this new Virtual DOM, the React diffing algorithm must compute the minimum number of operations needed to transform a tree into another and apply the mutation operations to the real DOM.

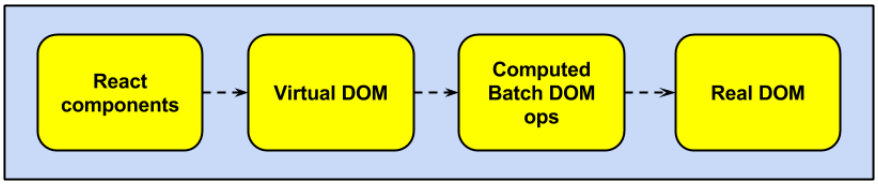

Figure 2.3: React DOM update

This diffing operation may seem time consuming and inefficient, considering that even some of the best algorithms for tree editing have a complexity of $O(n^3)$[1] (where n are the number of tree nodes). Therefore React implements a $O(n)$ algorithm using heuristics based on two assumptions[1]:

- We can assume that two components of the same class will generate similar trees and if they are of different classes they will generate different trees
- Developers can provide unique keys for the components that will not change between different renders so operations on lists of items can be done faster.

[1] https://facebook.github.io/react/docs/reconciliation.html

2.2.2 Bootstrap/Flexbox

For styling our HTML components we used the Bootstrap framework to achieve an experience similar to a real product. For laying out and aligning the panels that are auto-generated by exploring the Tastypie resource, we used the Flexbox module.

Bootstrap

Bootstrap is a powerful front-end framework created at Twitter by Mark Otto and Jacob Thornton[1], that features a rich collection of tools for faster development and quality work. The most appreciated features are the custom HTML and CSS components, plenty javascript plugins and a popular responsive page layout system. The project is also an open-source community backed project and it's main goal is to ease web development by giving the developers the ability to have consistent styling with minimum effort.

With Bootstrap we can simply structure the DOM with components like Panels by setting custom classes on the elements. An example can be seen in Listing 2.6 and the resulted component in Figure 2.4.

```
<div class="panel panel-default">
  <div class="panel-heading">
    <h3 class="panel-title">Panel title</h3>
  </div>
  <div class="panel-body">
    Panel content
  </div>
</div>
```

Listing 2.6: Panel with title

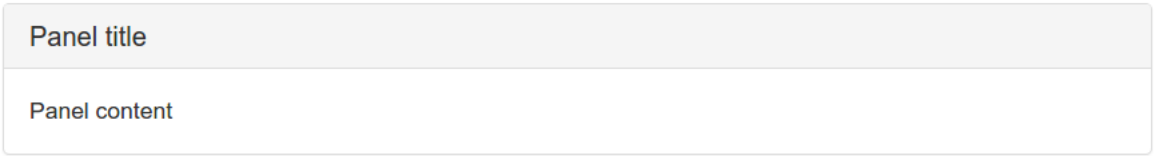

Figure 2.4: Panel with title

Flexbox

The Flexbox Layout (Flexible Box) module was built in the CSS3 standard, with the main goal of trying to properly alter width/height of items inside of a container, even if their sizes were unknown, hence the "flexible box" naming.

In the flex layout model, the children of a flex container can be laid out in any direction, and can "flex" their sizes, either growing to fill unused space or shrinking to avoid overflowing the parent[3]. An overview of the flex container can be seen in Figure 2.5.

We can see the main flex container being populated by flex items. In the flex layout the items will be laid out following either the `main axis` (from `main-start` to `main-end`) or the `cross axis` (from `cross-start` to `cross-end`).

[1] https://github.com/twbs/bootstrap

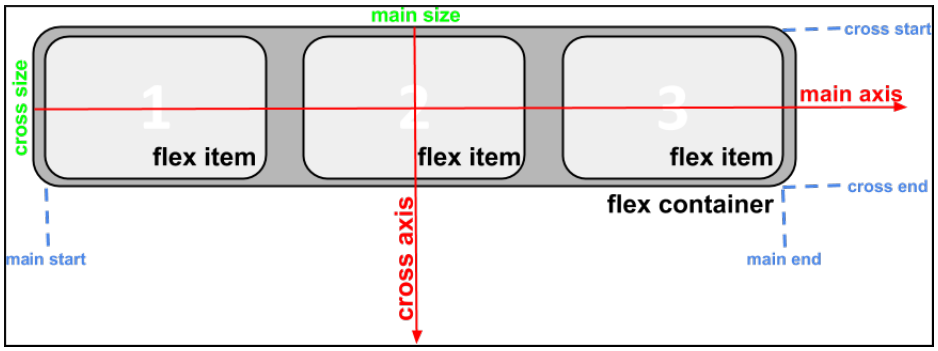

Figure 2.5: Flex container

The container has a collection of properties for laying out the items like: `flex-direction`, `flex-wrap`, `flex-flow`, `align-content`, `align-items`, `justify-content` which can be used for item layout and alignment. An example using some of theese can be seen in Listing 2.7 and Figure 2.6.

```
1 .div {
2   flex-flow: row-reverse wrap reverse;
3 }
```

Listing 2.7: Flexbox flex-flow usage

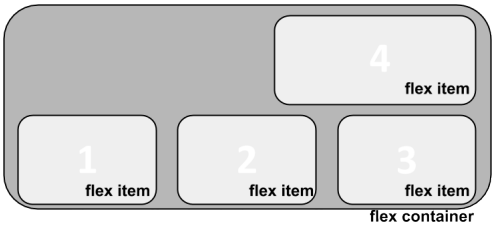

Figure 2.6: Flex-flow usage result

The `flex-flow` property is a shorthand for `flex-direction` and `flex-wrap` properties. It first establishes the main axis and direction of the layout (`row-reverse`), the opposite of inline direction and sets the items to wrap if they don't fit on the same row in the upwards direction (`upwards`).

Chapter 3

Implementation

3.1 Overview

3.2 Django models

3.2.1 API

3.2.2 App structure

3.3 React components

Chapter 4

Testing and evaluation

This is just a demo file. It should not be used as a sample for a thesis.

Chapter 5

Conclusions

This is just a demo file. It should not be used as a sample for a thesis.

5.1 Further development

Bibliography

[1] Philip Bille. A survey on tree edit distance and related problems.

[2] W3C Working Group. Web services architecture. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/, February 2006.

[3] W3C Working Group. Css flexible box layout module. http://www.w3.org/TR/css-flexbox-1/, May 2015.