

University POLITEHNICA of Bucharest

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



BACHELOR THESIS

Dynamic Webform Builder

Scientific Adviser:

ing. Andrei Picuş
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

Acknowledgements	i
Abstract	ii
1 Introduction	1
1.1 Objectives	1
1.2 Motivation	1
1.3 Background	2
1.3.1 Web Service	2
1.3.2 REST	3
1.3.3 HATEOAS	3
2 State of the Art	4
2.1 Back-end Frameworks	4
2.1.1 Django	4
2.1.2 Tastypie	6
2.2 Frontend-end Frameworks	8
2.2.1 React	8
2.2.2 Bootstrap/Flexbox	10
3 Implementation	13
3.1 Overview	13
3.2 Django Data Models	14
3.3 REST API	15
3.3.1 Resources	15
3.3.2 Data Output	16
3.4 React Components	17
3.4.1 Component Structure	17
3.4.2 Data Flow	18
3.4.3 Generic Form	19
4 Results	23
4.1 UI overview	23
4.2 Edit Panel	25
4.3 Resource exploration	26
5 Conclusions	28
5.1 Summary	28
5.2 Further developments	28
A Additional files	30
A.1 Post JSON Schema	30

A.2 Author JSON Schema	31
--	----

List of Figures

2.1	Django architecture	5
2.2	React architecture	9
2.3	React DOM update	10
2.4	Panel with title	11
2.5	Flex container	11
2.6	Flex-flow usage result	12
3.1	General application architecture	13
3.2	React components structure	18
3.3	Data flow inside components	19
3.4	GenericForm component	20
4.1	User interface structure	24
4.2	Edit panel states	25
4.3	DateTimeComponent in use	26
4.4	Resource exploration	27

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 describe the base concepts that are used across this thesis. Therefore we 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 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 moving¹ 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 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 developers¹ 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 is shown 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>

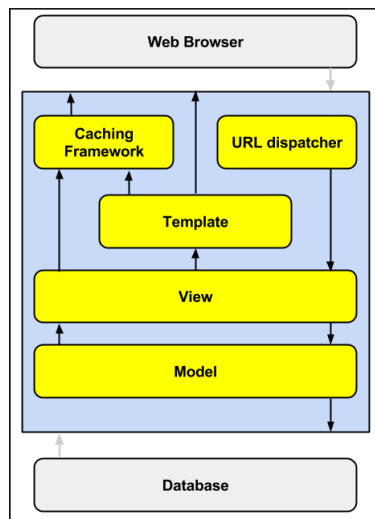


Figure 2.1: Django architecture

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. We present a very basic example of a Book model class in [Listing 2.1](#).

```

1 from django.db import models
2
3 class Book(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: Book model class

We observe that there are two character type fields and one date type field that map to the corresponding columns in [Listing 2.2](#).

```

1 CREATE TABLE book_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: Book 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 it is shown in [Listing 2.3](#).

```

1 <h1>{{ question.question_text }}</h1>
2 <ul>
3 {% for choice in question.choice_set.all %}
4     <li>{{ choice.choice_text }}</li>
5 {% endfor %}
6 </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 is presented in [Listing 2.4](#).

```

1 from django.conf.urls import include, url
2 from django.contrib import admin
3 from posts import views
4
5 urlpatterns = [
6     url(r'^$', views.index, name='index'),
7     url(r'^posts/', include('posts.urls')),
8     url(r'^admin/', include(admin.site.urls)),
9 ]
```

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

Tastypie¹ 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 behaviour 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.

JSON Schema

Another feature of Django Tastypie is its ability to also provide a JSON Schema of a specific resource. “JSON Schema defines the media type "application/schema+json", a JSON based format for defining the structure of JSON data. JSON Schema provides a contract for what JSON data is required for a given application and how to interact with it. JSON Schema is intended to define validation, documentation, hyperlink navigation, and interaction control of JSON data.”¹

In this project we use the JSON Schema mostly for understanding how the data is structured and act accordingly by generating the right components based on our data. A simple example of a JSON Schema usage is presented in [Listing 2.5](#). In this example we extracted a simple object from the API used in this project to better describe the purpose of the JSON Schema. Therefore the `title` object is set with with a `string` value. The corresponding Schema provides important data about this field like, its `type`, its `default` value or if the objects is editable trough the `readonly` property.

¹<http://json-schema.org/latest/json-schema-core.html>

```
1 Sample object
2
3 "title": "Demo_title"
4
5 JSON Schema
6 {
7   "blank": false,
8   "default": "No_default_provided.",
9   "help_text": "Title:",
10  "nullable": false,
11  "readonly": false,
12  "type": "string",
13  "unique": false
14 }
```

Listing 2.5: Simple JSON Schema

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 is presented in [Listing 2.6](#).

```
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.
11        handleChange}/>
12    );
13  });
14 React.render(<InputComponent val="Default_value" />, document.
  getElementById('content'));
```

¹<http://facebook.github.io/react/docs/why-react.html>

²<http://endot.org/notes/2014-01-30-react-rethinking-best-practices/>

Listing 2.6: 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 through `React.createClass` function. The most important elements of a React component are:

- **props** - Props are usually received from the parent component and should never be modified inside the component's functions as they are the component's 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 through `getInitialState()` function and through 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 is called JSX. The JSX syntax provides developers a more familiar way to describe the DOM elements that they deserve. In [Listing 2.6](#) the return value will be compiled by a JSX compiler to the equivalent call: `React.createElement('div', {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 explain how React is structured and how it uses these components to modify the DOM on user actions. In [Figure 2.2](#) we present a basic overview of the React architecture. The render cycle can be summarized in four steps:

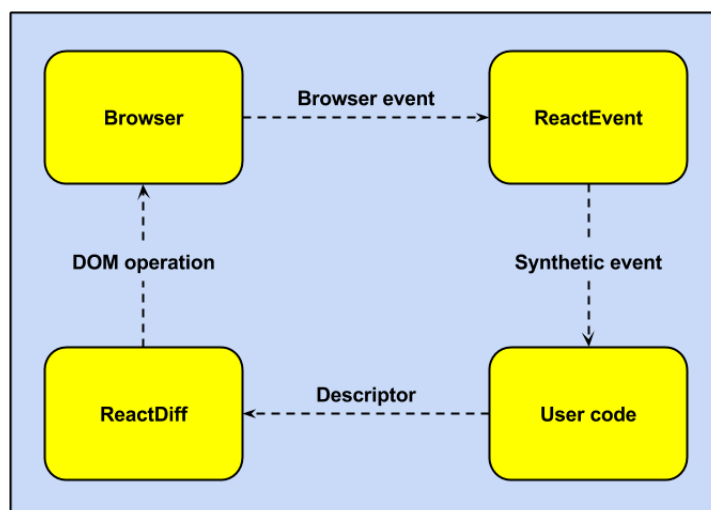


Figure 2.2: React architecture

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.
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 shown 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.

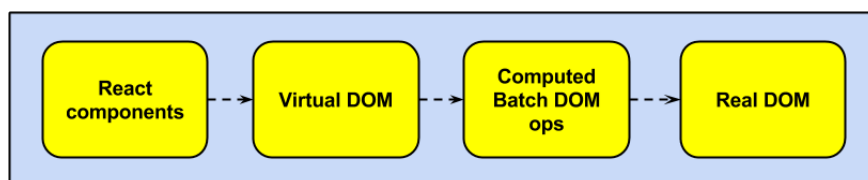


Figure 2.3: React DOM update

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 assumptions¹:

- 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.

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 Thornton², 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

¹<https://facebook.github.io/react/docs/reconciliation.html>

²<https://github.com/twbs/bootstrap>

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 is shown in [Listing 2.7](#) and the resulted component in [Figure 2.4](#).

```

1 <div class="panel_panel-default">
2   <div class="panel-heading">
3     <h3 class="panel-title">Panel title</h3>
4   </div>
5   <div class="panel-body">
6     Panel content
7   </div>
8 </div>

```

Listing 2.7: Panel with title

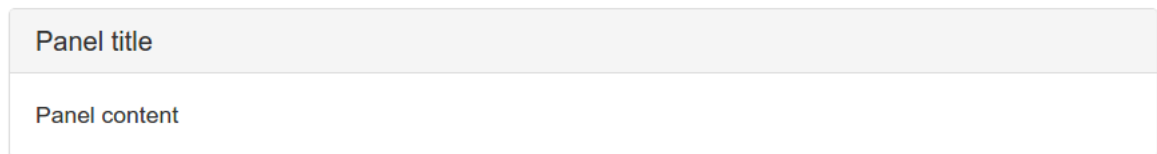


Figure 2.4: Panel with title

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 is presented in [Figure 2.5](#).

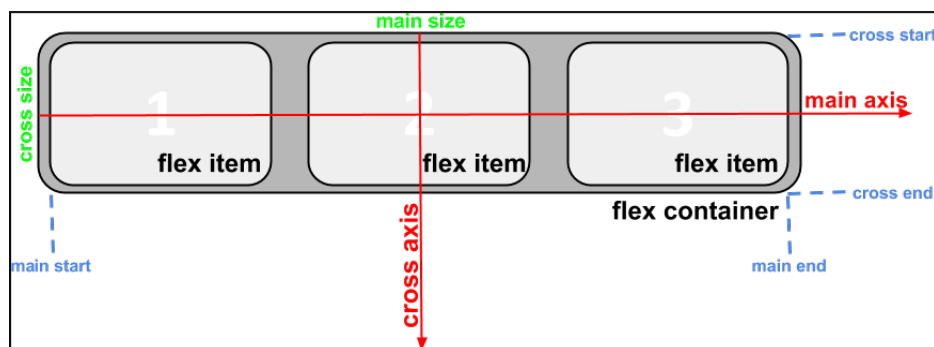


Figure 2.5: Flex container

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).

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 two of these is presented in [Listing 2.8](#) and [Figure 2.6](#).

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

Listing 2.8: Flexbox flex-flow usage

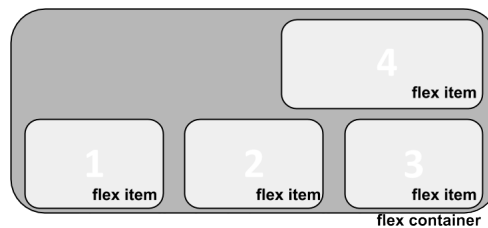


Figure 2.6: Flex-flow usage result

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

In this chapter we present the structure of our application and how all the technologies work together, starting with the [Overview 3.1](#) section.

Secondly we describe the back-end [Django Data Models 3.2](#) that we used to implement the REST API.

Then we present how Tastypie uses the Django data models to generate the API resource with its corresponding schema in section [REST API 3.3](#).

Finally in the [React Components 3.4](#) section, we present how these work together to explore the resource from the given API URI and how they auto-generate UI components based on the received JSON data.

3.1 Overview

The web application's goal is to demonstrate that an API can be explored and UI components can be auto-generated from the data that it receives. Therefore, for this to work, we established a Django based back-end that holds the requested or modified data and we set up a REST API using Tastypie. Next we implemented our front-end in React to explore the data received from the API and generate custom components. A general overview of all the application components and how they interact is presented in [Figure 3.1](#).

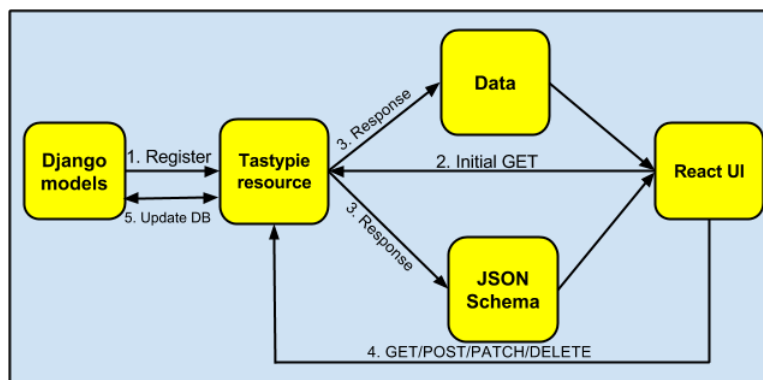


Figure 3.1: General application architecture

1. Register - First the Django models classes are defined ([Section 3.2](#)) and the SQL database tables are created and populated. After that a Django Tastypie Resource is declared using the previous defined models ([Section 3.3](#)) and the Register function is called using this resource to create the API.
2. Initial GET - The UI logic uses two AJAX calls (HTTP GETs) to retrieve data and schema about specified endpoint URI, with the purpose of rendering generic components.
3. Response - The Tastypie API queries the database from the Django server and returns responses to the previous two requests. The data returned is structured as a JSON and it describes the objects that are stored at that specific endpoint. Also along with the data the API returns the JSON Schema that describes the fields of the resource.
4. GET/POST/PATCH/DELETE - Finally the UI renders the data received from the initial GET and. Now the user can trigger browser events like deleting specific panels, adding new data or exploring and modifying existing data, thus sending the corresponding HTTP requests to the API for the updates.
5. Update DB - Once the API receives new requests from the UI it queries the database and updates it correspondingly through the hydrate/dehydrate cycle.

The next sections get into more detail on each of the steps described above.

3.2 Django Data Models

For the purpose of the project we only needed a functional REST API and a way to render the main HTML page. The first step of establishing the REST API was to define the Django data models. For testing purposes we defined a `Post` entity, similar to a usual blog post, with the following properties:

- `title` - The title of the post.
- `author` - A related entity that describes the data about the author of the specific post.
- `created_at` - A field that describes the date and time the post was created at.
- `content` - The actual text of the post.

The `author` field is important for the purpose of demonstrating the capability to explore related fields and describe them. The `author` data model is actually inherited from Django's `contrib.auth.models` `User` model which is composed of standard fields as: `username`, `email`, `first name`, `last name`, `date joined` and also fields that we chose to exclude through filters like: `password`, `is_active`, `is_staff`, `is_superuser`.

In [Listing 3.1](#) we present the Django `Post` model class implementation. As all Django model classes, it inherits the `models.Model` class and its standard field types.

```

1 from django.db import models
2 from django.contrib.auth.models import User
3
4 class Post(models.Model):
5     title = models.CharField(max_length=200, help_text='Title:␣')
6     author = models.ForeignKey(User, help_text='User')
7     created_at = models.DateTimeField(auto_now_add=True, help_text='
        Created_at:␣')
8     content = models.TextField(help_text='Content:␣')
```

Listing 3.1: Post model class

Field types are classes that describe and encapsulate certain types of data. As it is presented in the listing above field types can hold additional parameters as `help_text`, which is used to provide additional information about the field and subsequently displayed in the REST API's JSON Schema. Another optional parameter is `DateTimeField`'s `auto_now_add` which is set to `True` to automatically set the field to now when the object is first created.

3.3 REST API

The REST API set up through Django Tastypie is the most important part of our back-end. It takes the data models described in the previous section and outputs them in a RESTful way. We will next present how we set up the API and also its resulted output.

3.3.1 Resources

Previously we described the data models for a generic blog post and its related author model. Based on those models we created API resources that are able to expose that data.

Post Resource

In [Listing 3.2](#) we present the implementation for the `PostResource` resource. The resource inherits the base methods from the `ModelResource` class that itself is a subclass of `Resource`, designed to work with Django's Models.

```
1 class PostResource(ModelResource):
2
3     author = fields.ForeignKey(UserResource, 'author')
4
5     class Meta:
6         queryset = Post.objects.all()
7         resource_name = 'post'
8         authorization = Authorization()
9         always_return_data = True
10
11     def build_schema(self):
12         base_schema = super(ModelResource, self).build_schema()
13         base_schema['fields']['author']['
14             'resource'] = this.author.get_resource_uri()
15         return base_schema
```

Listing 3.2: Post resource

The inner `Meta` allows for class-level configuration of how the `Resource` should behave. Therefore we can set parameters like `resource_name`, `authorization` and the most important `queryset`. The `queryset` provides the resource with the set of Django models to respond with. Also because the `author` field is a related data model we want to provide it with a `ForeignKey` to the `UserResource` to provide a link to the related URI. A more detail output of the API will be presented in

In case of a related component the JSON Schema should describe the URI of the parent resource. For this, as seen in lines 11-14 in the above listing, we overwrote the `build_schema` method in the `Post` resource class to add another field called `resource` that would expose the author resource URI.

User Resource

The UserResource follows the same design that we presented in the section before, on the PostResource. While it will not have a modified `build_schema` method it will have some additional Meta fields. [Listing 3.3](#) shows only what changed in the Meta class. For security reasons we excluded the fields like `password`, `is_active`, `is_staff`, `is_superuser` through the `excludes` option. The filtering option provides the resource with a list of fields that will accept client filtering on. In this case by setting the `'username': ALL` option we can now use specific queries like `api/v1/author/?username=flaviusone&format=json`.

```

1 class Meta:
2     excludes = [
3         'password', 'is_active', 'is_staff', 'is_superuser']
4     filtering = {
5         'username': ALL,
6     }

```

Listing 3.3: User resource

After setting up the resources we continued by setting up the API for the URL dispatcher in Django's `urls.py` file. Next, we instantiated the resources and we called the Tastypie provided constructor `Api(api_name='v1')` with a parameter for setting an API name. In the end we called the `register()` method on the API object with both resources as parameters and we added the `url()` call in the `urlpatterns` array as follows `url(r'^api/', include(v1_api.urls))`

3.3.2 Data Output

With the API properly setup we now have output on the resource specific URI, in our case `/posts/api/v1/post/?format=json`. In [Listing 3.4](#) we present a sample output of the Post resource (only one object displayed).

```

1 {
2     "meta": {
3         "limit": 20,
4         "next": null,
5         "offset": 0,
6         "previous": null,
7         "total_count": 1
8     },
9     "objects": [
10        {
11            "author": "/posts/api/v1/author/3/",
12            "content": "This_post_has_demo_content",
13            "created_at": "2015-06-11T18:18:00",
14            "id": 1,
15            "resource_uri": "/posts/api/v1/post/3/",
16            "title": "This_is_a_demo_post"
17        }
18    ]
19 }

```

Listing 3.4: Post resource data output

The listing above shows the return of a single object with two properties: `meta` and `objects`. The `meta` object provides info about the number of objects displayed by in the response. Some important parameters here are `limit` and `next` which describe how many objects can be returned in one response and what is the path for the next page of objects in case the response is too big and the system automatically splits the response into separate calls to reduce bandwidth.

The `objects` property describes an array of objects that are located at the specified resource. The data is also described in JSON format, but it can also be formatted as XML by using the `?format=xml` in the request call. In the case of the `author` field the API specifies the related path to the location of the data by obeying HATEOAS principles. By further exploring this URI we get the following additional JSON object, presented in [Listing 3.5](#), that further describes the author object.

```
1 {
2   "date_joined": "2015-06-11T00:23:00",
3   "email": "flaviusone@gmail.com",
4   "first_name": "Flavius",
5   "id": 3,
6   "last_login": "2015-06-11T00:24:00",
7   "last_name": "Tirnacop",
8   "resource_uri": "/posts/api/v1/author/3/",
9   "username": "flaviusone"
10 }
```

Listing 3.5: Author entry

Besides the data output of the `Post` resource we can also access the JSON Schema available at `/posts/api/v1/post/schema/?format=json`. The output of this call can be found in [Appendix A.1](#). The JSON Schema describes the type of each field that is used by the front-end to generate UI components. The standard types described by the Schema can be `string`, `integer`, `datetime` and `related`. In the case of a `related` type object the front-end logic will explore the additional `resource` parameter and display the available data. Another important field used by the front-end logic is the `readonly` attribute, that points if the object field is editable.

3.4 React Components

As we presented in the [Overview 3.1](#) section, the main element of the project is the reactive UI. The interface is built to be agnostic of any changes in the back-end by generating specific components based on the data provided and the JSON Schema. In this section we will present how the React components are structured and how they work together.

3.4.1 Component Structure

By its nature, a web application built in React is composed of multiple reusable React Components. In [Figure 3.4](#) we present a simplified version of the UI component structure that describes the following elements:

- `<FormBox/>` - This is the main component that encapsulates all other. This component also has the main purpose of calling the AJAX requests that are called by other components that are its children.

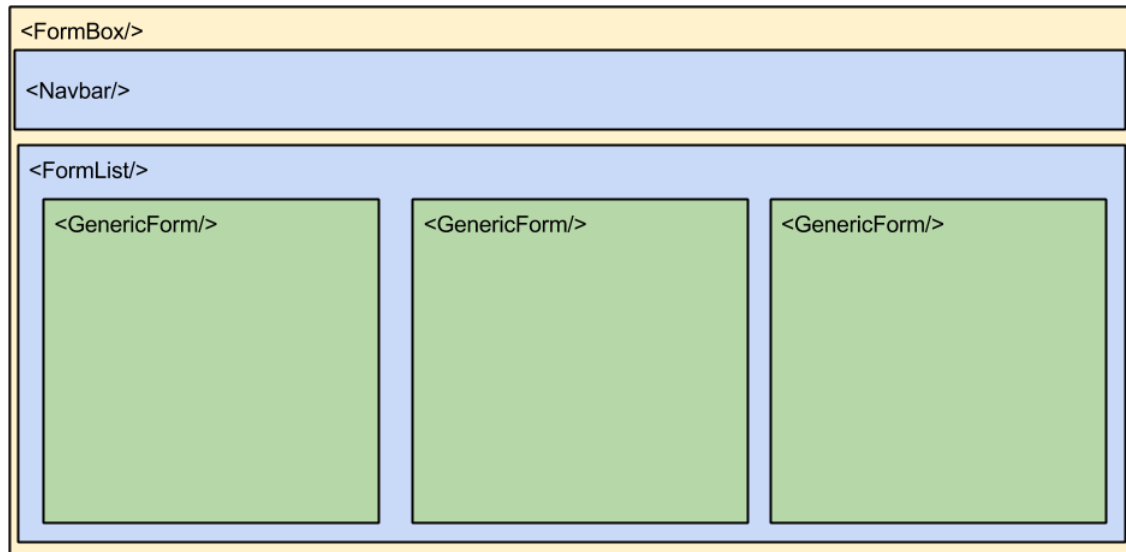


Figure 3.2: React components structure

- `<Navbar/>` - This is not in fact a specific React component but its a structure composed of all the elements that form the page's navigation bar that are actually react components imported from the `react-bootstrap` package. Some of the components used include:
 - `<Navbar/>` and `<Nav/>` components, used for encapsulating the URL input form.
 - `<Input/>` component that describes a simple input form for the URL options.
 - `<ButtonInput/>` component that describes the submit button for the input form described above
- `<FormList/>` - This component receives an array of objects from its parent and generates and encapsulates the `<GenericForm/>` components.
- `<GenericForm/>` - This represents the main component that the user interacts with. It also encapsulates components for each type of object. We will continue to present this component in further detail in [Section 3.4.3](#).

3.4.2 Data Flow

We presented in [Section 2.2.1](#) the data flow within the React framework and we explained how components have states and properties as the main mechanism of storing and transmitting data. Next, we will describe how data flows in our application from the API through the components and back. Based on the simplified diagram is presented in [Figure 3.3](#) we can summarize the process in the following steps:

1. The `FormBox` component is the center piece of communication with the REST API. On the initial render this component calls an AJAX GET to retrieve data and schema from the resource URI. The data received is structured as an object array as presented in [Section 3.3.2](#)
2. Next the objects array and schema are passed to the `FormList` upon its rendering as properties.
3. Once the `FormList` `render()` function is called, it takes the received data and for each object it renders a `GenericForm`. The generated `GenericForm` components will also

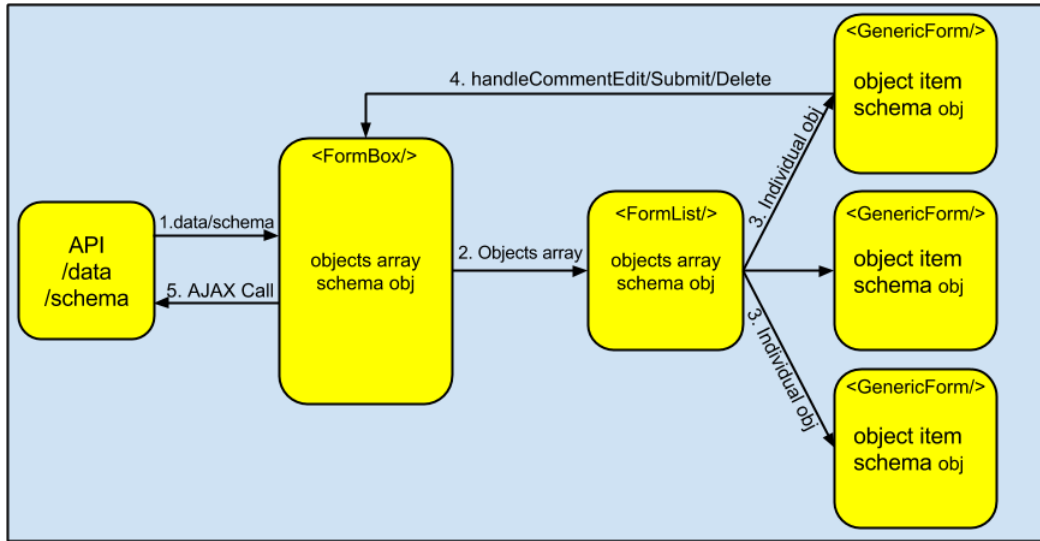


Figure 3.3: Data flow inside components

receive the schema along with the object data in its proprieties.

4. With the received data, the `GenericForm` components can now render the corresponding elements based on the schema object. After the `GenericForm` component is rendered it can now be added, deleted or edited by the user. When the user presses the submit button on the component, the new data is sent trough a callback that can look like `handleCommentEdit`. This callback is also inherited trough props from the component's parent. in this case `FormList`, that also inherits the callback from its parent `FormBox`. Therefore in the end the new data arrives at the `FormBox` component.
5. The last step consists of the `FormBox` component to trigger the AJAX call with the new data received from its children. The AJAX call can be of different types:
 - (a) POST - If a new object is added trough a modified `GenericForm` that describes a form with empty fields that acts as an `AddForm`
 - (b) DELETE - If a form is deleted by the user.
 - (c) PATCH - If a form component changes its values and triggers and edit action.

3.4.3 Generic Form

The `GenericForm` is the core component of the reactive UI. Trough its behavior the UI can auto-generate components without having a preset template from the beginning. Also trough the `GenericForm` component the UI can auto-explore and re-render further discovered data.

For the component to be able to accommodate any type of data several auxiliary components have been created. In [Figure 3.4](#) we present the internal structure of the `GenericForm` component.

The figure shows four different components that describe the four available types in our API (`string`, `integer`, `datetime`, `related`). It is important to mention that these four types of data are not the only ones possible. We could also have components that describe boolean data or enumeration data. Next, we will continue to describe the structure of the four data types.

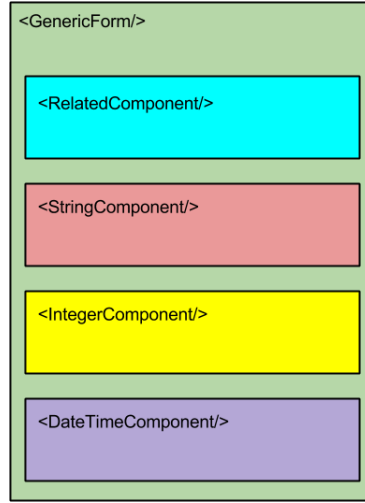


Figure 3.4: GenericForm component

The `GenericForm`'s render function follows a simple algorithm described in [Algorithm 1](#). It has the main purpose to return a `div` container populated with content. This content is actually an array of HTML elements returned by the specific components that form the `GenericForm` object. To initialize the type specific components the algorithm takes each property and searches for its type in the general schema object. Once the type is known the proper component is initialized by receiving additional data as React props. These props include the actual `key:value` property, the schema object (used mainly for related objects and also for readonly objects) and the `display_state`.

The `display_state` is a `GenericForm` state that can have two values: `show` or `edit`. This state describes if the component can be edited by the user. When a user presses the component Edit button the state changes from its default `show` value to `edit` and as any `setState` function call triggers a component re-render. Upon the new render the `GenericForm` component will now send the new `display_state` for its children that will change their state accordingly.

```

1 content ← []
2 foreach property in object do
3   type ← type of property extracted from the schema object
4   switch type do
5     case string
6       content.push(StringComponent(property, schema, display_state))
7     end
8     case integer
9       content.push(IntegerComponent(property, schema, display_state))
10    end
11    case datetime
12      content.push(DateTimeComponent(property, schema, display_state))
13    end
14    case related
15      content.push(RelatedComponent(property, schema, display_state))
16    end
17  end
18 end
19 return content

```

Algorithm 1: GenericForm render algorithm

As we described until now the `GenericForm` component is a container for data components. We will now follow to present the four types of React components that we implemented within our project.

StringComponent

The `StringComponent` is a generic React component that has the role of displaying string type objects. The component also has two states `show` and `edit` that facilitate the user to edit it by typing into an editable `textarea`. Having a `textarea` input the component must store the current data into its state. Then, the `textarea` as an `onChange={this.handleChange}` method applied on it that on a change event calls the `handleChange` function. This function has the role of setting the new state of the value object for the `StringComponent`.

IntegerComponent

The `IntegerComponent` has a similar structure to the `StringComponent`. In our project this component is used to display the `posts` or `authors` specific `ids`.

DateTimeComponent

The `DateTimeComponent` is a more complex component than the `StringComponent` or `IntegerComponent`. It receives from its parent through its props a raw date that is usually formatted as follows `2015-06-11T14:09:00`. It then takes this value and converts it into Coordinated Universal Time (UTC) and inputs it to a Bootstrap Datepicker component. The Bootstrap Datepicker component is an open source project that can be found at ¹ and has the role of providing an aesthetic and user friendly interface to choose a date and time for the specific components.

RelatedComponent

The `RelatedComponent` is the most special and important component because it helps demonstrate that auto-exploration of an API and further generation of components can be achieved. The component initially receives the URI of the related component and also the schema object. A simplification of the algorithm behind the `RelatedComponent` is presented in Algorithm 2.

```

1 mainResource ← schema.resource
2 loadDataIntoDropdown(mainResource)
3 if edit is pressed then
4   explore ← true
5   exploreURI ← selected dropdown data
6   newData ← getDataFromServer(exploreURI)
7   content ← new GenericForm(newData)
8 end
9 return content

```

Algorithm 2: `RelatedComponent` render algorithm

¹ <http://eonasdan.github.io/bootstrap-datetimepicker/>

The first step is getting the URI from the resource field in the schema. As an example this URI may look like `/posts/api/v1/author/` and point to the location of all the elements that are available at the resource where the related component URI (eg. `/posts/api/v1/post/1/`) came from. With this data the component can now load it into a dropdown for the user to select if he wants which object to explore. After the desired resource was selected in the dropdown, when the user presses the `Edit` button the state changes from `explore:false` to `explore:true` and the component calls an AJAX GET request to retrieve the data (and schema) about that certain object. With the new data the component now initializes another `GenericForm` component that will render the new data as described in the previous sections.

Chapter 4

Results

In this chapter we present how the front-end application works from the user's standpoint and how the React components mutate upon browser events. In section [UI overview 4.1](#) we present how the UI looks and how its structured with the rendered components.

Next, in section [Edit Panel 4.2](#) we describe how the `GenericForm` components generate edit panels and how they are displayed.

Finally in the [Panel Nesting 4.3](#) section we describe how the application renders the explored content through the API.

4.1 UI overview

In [Figure 4.1](#) we present the rendered user interface and how the React components are structured after they are rendered using the Bootstrap framework and Flex CSS. It can also be seen that the parent component is the `FormBox` that encapsulates the other UI elements.

In the `Navbar` component the user uses a text input to pick or insert any API endpoint URI and then he can press the render button to trigger the initial AJAX GET call.

On the server response the `FormList` is initialized with `GenericForm` components. In the example presented below we can see a list of six `GenericForm` components. Even if the first component looks different from the other five, the `AddForm` panel is in fact a `GenericForm`, which we will present in more detail in subsection [Add Panel 4.2](#).

The other `GenericForm` components describe a simple usage of an admin panel for a list of `Posts`. We can see that the post components feature info about the `title`, `content`, `author`, `id` and `creation_date`. The panels also feature buttons for user interaction like editing or deleting. Next we will present the behavior of the post panels on user interaction and we will describe how the data inside is structured.

Dynamic Form Builder Version 0.4

/posts/api/v1/post/

Render

<Navbar/>

Add Form

Author : /posts/api/v1/author/1/

Content :

Created At:

Id :

Resource Uri : /posts/api/v1/post/

Title :

Submit

<GenericForm/>

Edit show

Delete

Author : /posts/api/v1/author/1/

Content : "Everybody knows a certain thing is unrealizable until somebody unaware of this comes and invents it." Albert Einstein

Created At: Fri, 12 Jun 2015 06:02:00 GMT

Id : 5

Resource Uri : /posts/api/v1/post/5/

Title : Favorite quote 1

<GenericForm/>

Edit show

Delete

Author : /posts/api/v1/author/2/

Content : "Your work is going to fill a large part of your life, and the only way to be truly satisfied is to do what you believe is great work. And the only way to do great work is to love what you do. If you haven't found it yet, keep looking. Don't settle. As with all matters of the heart, you'll know when you find it." Steve Jobs

Created At: Fri, 12 Jun 2015 09:02:00 GMT

Id : 4

Resource Uri : /posts/api/v1/post/4/

Title : Favorite quote 2

<GenericForm/>

Edit show

Delete

Author : /posts/api/v1/author/3/

Content : "You have to learn the rules of the game. And then you have to play better than anyone else." Albert Einstein

Created At: Thu, 11 Jun 2015 11:09:00 GMT

Id : 3

Resource Uri : /posts/api/v1/post/3/

Title : Favorite quote 3

<GenericForm/>

Edit show

Delete

Author : /posts/api/v1/author/3/

Content : "Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning." Albert Einstein

Created At: Mon, 08 Jun 2015 06:24:00 GMT

Id : 2

Resource Uri : /posts/api/v1/post/2/

Title : Favorite quote 4

<GenericForm/>

Edit show

Delete

Author : /posts/api/v1/author/1/

Content : "In matters of truth and justice, there is no difference between large and small problems, for issues concerning the treatment of people are all the same." Albert Einstein

Created At: Thu, 11 Jun 2015 09:18:00 GMT

Id : 1

Resource Uri : /posts/api/v1/post/1/

Title : Favorite quote 4

<FormList/>

<FormBox/>

Figure 4.1: User interface structure

4.2 Edit Panel

The `EditPanel` is the main component of the UI. It renders the data that is retrieved from the API regardless of its type. As described in the [GenericForm 3.4.3](#) section, the panel will have two states: `show` and `edit`. In [Figure 4.2](#) we present how does the `EditPanel` mutate when the user triggers a state change from `show` (default) to `edit` by pressing the `Edit` `show` button.

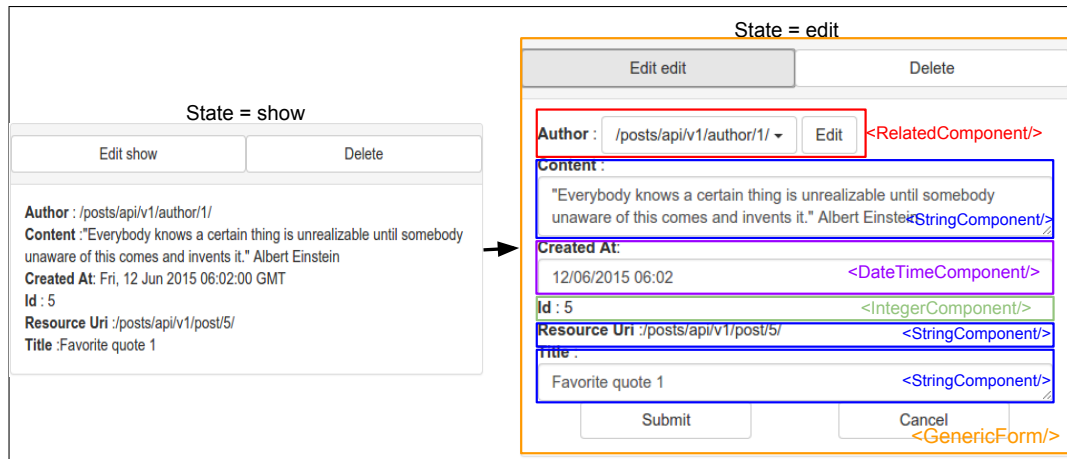
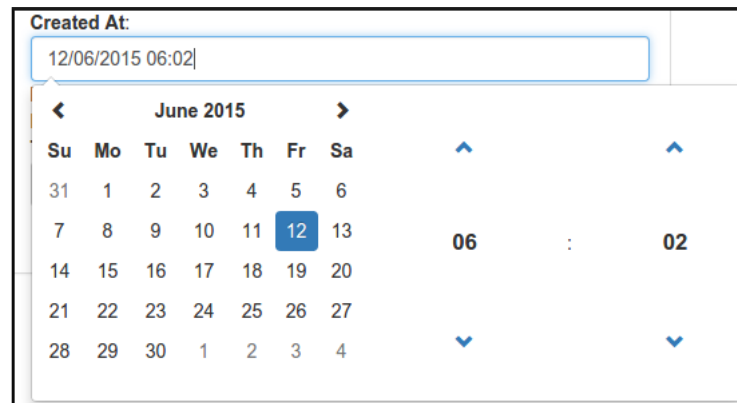


Figure 4.2: Edit panel states

In the right section of the figure above we describe the structure of the `GenericForm` component. In the `edit` state all components have a specific behavior described by the auto-generated html elements based on the `key:value` structured object properties as follows:

- The **RelatedComponent** (in red) is structured as a string key (`Author`) followed by a dropdown component and an `Edit` button. When the `GenericForm` component changes state from `show` to `edit` the dropdown component is populated with data that describes the key object (in our case the `Author`). Finally, the `Edit` button triggers the `exploration` method in the `RelatedComponent` which we will present in more detail in the [Panel Nesting 4.3](#) section.
- The **StringComponent** (in blue) appears multiple times and is represented by a string key (eg. `Content`) and a `textarea` HTML component. Also it can be seen that there are cases when the component will be `readonly` (in our case `Resource URI`) therefore the value will only be displayed as a string.
- The **IntegerComponent** (in green) has a similar structure with the `StringComponent` and in our case it also features a `readonly` mode.
- Finally the **DateTimeComponent** is composed of a string key (`Created At`) and a field that displays the date and time and on a `onClick` event initializes the `Bootstrap Datetimepicker`¹ component which is shown in [Figure 4.3](#). The component adds a consistent look and experience to the application which is one of the main objectives of this project.

¹ <http://eonasdan.github.io/bootstrap-datetimepicker/>



The image shows a web form component titled "Created At:". Below the title is a text input field containing "12/06/2015 06:02". Below the input field is a date and time picker. The date picker shows a calendar for June 2015, with the 12th of June selected. The time picker shows "06" for the hour and "02" for the minutes, separated by a colon. There are up and down arrows for navigating between months and hours/minutes.

Figure 4.3: DateTimeComponent in use

Add Panel

The `AddPanel` component (also shown in [Figure 4.1](#)) can be used by the user to submit a new `Post` entity to the database through the REST API. This component is actually a `GenericForm` that receives an optional property from the `FormList` parent. On initialization with the optional property the component will not render the `Edit`, `Delete` and `Cancel` buttons. Also the `Add` component must receive an empty object from its parent to be able to render empty input components.

4.3 Resource exploration

`RelatedComponents` have the ability to explore the given URI path and auto-generate more UI components based on this action. As presented in the [Generic Form 3.4.3](#) section the `RelatedComponent`, when a user presses the `Edit` button it triggers an AJAX GET call on that specific resource. Once the data is retrieved it will generate another `GenericForm` component that has all the properties of a normal one. This specific behavior is presented in [Figure 4.4](#).

The diagram illustrates the process of resource exploration in a REST client. It shows two forms side-by-side, connected by a right-pointing arrow. The left form, titled "Explore = False", is a basic form for editing a resource. It includes fields for "Author" (a dropdown menu), "Content" (a text area), "Created At" (a date field), "Id" (a text field), "Resource Uri" (a text field), and "Title" (a text field). The right form, titled "Explore = True", is a more detailed form for editing a resource. It includes fields for "Author" (a dropdown menu), "Content" (a text area), "Created At" (a date field), "Id" (a text field), "Resource Uri" (a text field), and "Title" (a text field). Additionally, it includes a "Date Joined" field, an "Email" field, a "First Name" field, a "Last Name" field, a "Last Login" field, and a "Username" field. Both forms have "Edit" and "Delete" buttons at the top. The right form also has an "Edit show" button. The right form is labeled with "<GenericForm/>" at the top right and bottom right. The left form is labeled with "<GenericForm/>" at the top right.

Explore = False

Author : /posts/api/v1/author/3/ Edit <RelatedComponent/>

Content :

Created At:

Id : 5

Resource Uri : /posts/api/v1/post/5/

Title :

Favorite quote 1

Submit Cancel

Explore = True

Author : /posts/api/v1/author/3/ Edit <RelatedComponent/>

Date Joined: Thu, 11 Jun 2015 00:23:00 GMT

Email : flaviusone@gmail.com

First Name : Flavius

Id : 3

Last Login: Thu, 11 Jun 2015 00:24:00 GMT

Last Name : Tirmacop

Resource Uri : /posts/api/v1/author/3/

Username : flaviusone

Content :

Created At:

Id : 5

Resource Uri : /posts/api/v1/post/5/

Title :

Favorite quote 1

Submit Cancel

Figure 4.4: Resource exploration

Chapter 5

Conclusions

In this chapter we conclude our description on this project. In the first section [Summary 5.1](#) we'll do a summary of how and what we managed to prove in this paper.

In the second section, [Further development 5.2](#) we will discuss what could be developed next, starting from our work described in the [Implementation 3](#) chapter.

5.1 Summary

In the first section of the opening chapter of this, thesis we pointed out the main goals that the project wanted to achieve and that we aimed to demonstrate if it is possible to have auto-exploration and auto-generation in a back-end agnostic front-end application. Based on these ideas we will present what we managed to achieve at the end of the development period.

- Set up a back-end that features Django data models that are used for exposing data in a RESTful way.
- Set up a REST API and model schema in a single place (server-side) using Django Tastypie framework.
- Set up a consistent looking UI that is able to auto-explore a given resource and auto-generate generic forms.
- Customizing the generic forms for the user to be able to edit/delete or add them with minimal client-side work.
- Laid down the base for a scalable user interface trough styling frameworks like Bootstrap and Flexbox.

I consider developing an application that requires knowledge of the full development stack (back-end and front-end) as a big accomplishment for me and for my experience as a software engineer. By having to work with state of the art technologies like React, usually not used or taught at the university courses, I had to first learn how their architecture works and only then apply them to my project.

5.2 Further developments

The purpose of the project was only to demonstrate that it is possible to build a front-end that is agnostic of it's back-end. Thus the relations between the data models we constructed were

mostly **one-to-many** in the case of an author and his related posts. While this helps us to prove our point and have our front-end explore through the post resource, the author that is assigned to it, we can encounter more complex cases in real life that.

The next step in our implementation would be integrating **many-to-many** relations between the data entities. A concrete example in our implementation would be having multiple authors assigned to multiple posts and being able to edit the authors assigned to a post accordingly.

Even if the work can be still improved by adding new relationship models or new generic type components we could pack the front-end application to work as a generic framework and later open source it to the community for further improvements.

Appendix A

Additional files

A.1 Post JSON Schema

```
1 {
2   "allowed_detail_http_methods": [
3     "get",
4     "post",
5     "put",
6     "delete",
7     "patch"
8   ],
9   "allowed_list_http_methods": [
10    "get",
11    "post",
12    "put",
13    "delete",
14    "patch"
15  ],
16  "default_format": "application/json",
17  "default_limit": 20,
18  "fields": {
19    "author": {
20      "blank": false,
21      "default": "No_default_provided.",
22      "help_text": "A_single_related_resource._Can_be_either_a_URI
23                  _or_set_of_nested_resource_data.",
24      "nullable": false,
25      "readonly": false,
26      "related_type": "to_one",
27      "resource": "/posts/api/v1/author/",
28      "type": "related",
29      "unique": false
30    },
31    "content": {
32      "blank": false,
33      "default": "",
34      "help_text": "Content:_",
35      "nullable": false,
```

```

35         "readonly":false,
36         "type":"string",
37         "unique":false
38     },
39     "created_at":{
40         "blank":true,
41         "default":true,
42         "help_text":"Created_at:_",
43         "nullable":false,
44         "readonly":false,
45         "type":"datetime",
46         "unique":false
47     },
48     "id":{
49         "blank":true,
50         "default":"",
51         "help_text":"Integer_data._Ex:_2673",
52         "nullable":false,
53         "readonly":false,
54         "type":"integer",
55         "unique":true
56     },
57     "resource_uri":{
58         "blank":false,
59         "default":"No_default_provided.",
60         "help_text":"Unicode_string_data._Ex:_\"Hello_World\"",
61         "nullable":false,
62         "readonly":true,
63         "type":"string",
64         "unique":false
65     },
66     "title":{
67         "blank":false,
68         "default":"No_default_provided.",
69         "help_text":"Title:_",
70         "nullable":false,
71         "readonly":false,
72         "type":"string",
73         "unique":false
74     }
75 }
76 }

```

Listing A.1: Post resource schema

A.2 Author JSON Schema

```

1 {
2     "allowed_detail_http_methods":[
3         "get",
4         "post",
5         "put",

```

```

6         "delete",
7         "patch"
8     ],
9     "allowed_list_http_methods": [
10        "get",
11        "post",
12        "put",
13        "delete",
14        "patch"
15    ],
16    "default_format": "application/json",
17    "default_limit": 20,
18    "fields": {
19        "date_joined": {
20            "blank": false,
21            "default": "2015-06-20T14:49:00.629417",
22            "help_text": "A_date_&_time_as_a_string._Ex:_\"2010-11-10T03
                :07:43\"",
23            "nullable": false,
24            "readonly": false,
25            "type": "datetime",
26            "unique": false
27        },
28        "email": {
29            "blank": true,
30            "default": "",
31            "help_text": "Unicode_string_data._Ex:_\"Hello_World\"",
32            "nullable": false,
33            "readonly": false,
34            "type": "string",
35            "unique": false
36        },
37        "first_name": {
38            "blank": true,
39            "default": "",
40            "help_text": "Unicode_string_data._Ex:_\"Hello_World\"",
41            "nullable": false,
42            "readonly": false,
43            "type": "string",
44            "unique": false
45        },
46        "id": {
47            "blank": true,
48            "default": "",
49            "help_text": "Integer_data._Ex:_2673",
50            "nullable": false,
51            "readonly": false,
52            "type": "integer",
53            "unique": true
54        },
55        "last_login": {
56            "blank": false,
57            "default": "2015-06-20T14:49:00.629387",

```

```

58         "help_text": "A_date_&_time_as_a_string._Ex:_\"2010-11-10T03
59             :07:43\\",
60         "nullable": false,
61         "readonly": false,
62         "type": "datetime",
63         "unique": false
64     },
65     "last_name": {
66         "blank": true,
67         "default": "",
68         "help_text": "Unicode_string_data._Ex:_\"Hello_World\\",
69         "nullable": false,
70         "readonly": false,
71         "type": "string",
72         "unique": false
73     },
74     "resource_uri": {
75         "blank": false,
76         "default": "No_default_provided.",
77         "help_text": "Unicode_string_data._Ex:_\"Hello_World\\",
78         "nullable": false,
79         "readonly": true,
80         "type": "string",
81         "unique": false
82     },
83     "username": {
84         "blank": false,
85         "default": "No_default_provided.",
86         "help_text": "Required._30_characters_or_fewer._Letters,_
87             digits_and_@/./+/-/_only.",
88         "nullable": false,
89         "readonly": false,
90         "type": "string",
91         "unique": true
92     }
93 },
94 "filtering": {
95     "username": 1
96 }

```

Listing A.2: Post resource schema

Bibliography

- [1] Philip Bille. A survey on tree edit distance and related problems, December 2004.
- [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.
- [4] MyTARDIS. Django architecture. <http://mytardis.readthedocs.org/en/latest/architecture.html#functional-architecture>, 2013.