Framework for Web application development based on Java technologies and AngularJS

Lazar Nikolić, Gordana Milosavljević, Igor Dejanović
University of Novi Sad, Faculty of Technical Sciences, Serbia
{lazar.nikolic, grist, igord}@uns.ac.rs

***Abstract* – The paper presents a framework for developing single page Web applications based on Java technologies and AngularJS. Unlike traditional code-generation, the frontend of this solution uses a set of built-in generic user interface components capable of adapting to any metadata provided from the backend. The goal is to quickly provide a base for building a fully functional application and to switch focus from frontend development to modeling and backend development.**

I. INTRODUCTION

Persistent (data) model of an application usually defines concepts, types, constraints and relationships needed for the application functioning. This model is traditionally used for automatic generation of a persistent layer of the application. But, this model, enhanced with some presentation details can also be used for automatic construction of a presentation layer, in order to minimize the effort needed for its development.

In contrast with solutions presented in [3, 4, 5] that require manual coding or frontend generation based on a persistent model, our approach uses a reusable AngularJS client framework that adapts itself according to the metadata extracted from persistence layer (figure 1). This can greatly reduce the time needed to develop an initial version of the fully functioning application.

Architecture of the framework consists of four layers (figure 1):

- **Persistence layer** – contains class definitions, O/R mappers and provides data manipulation.
- **Service layer** – contains RESTful services for client-server communication.
- **Transformation layer** – contains mechanisms for object serialization and metadata extraction.
- **Presentation layer** – AngularJS client application.

The first three layers form the backend, while Presentation layer forms the frontend. Each part of the architecture will be briefly described in the next chapters.

Although concepts used for building the framework are platform independent, the framework itself is implemented using Java technologies for the backend and AngularJS for the frontend.

II. RELATED WORK

Rapid application development is currently an active field of research. Kroki [11, 12, 13] is a rapid prototyping tool that allows users to actively participate in application development. It uses a Java based application engine to quickly create an executable prototype. However, it does not make use of up-to-date technologies, such as AngularJS.

Our framework uses EJB components at the backend, similar to work presented in [9]. In [9] is presented implementation of intermediate form

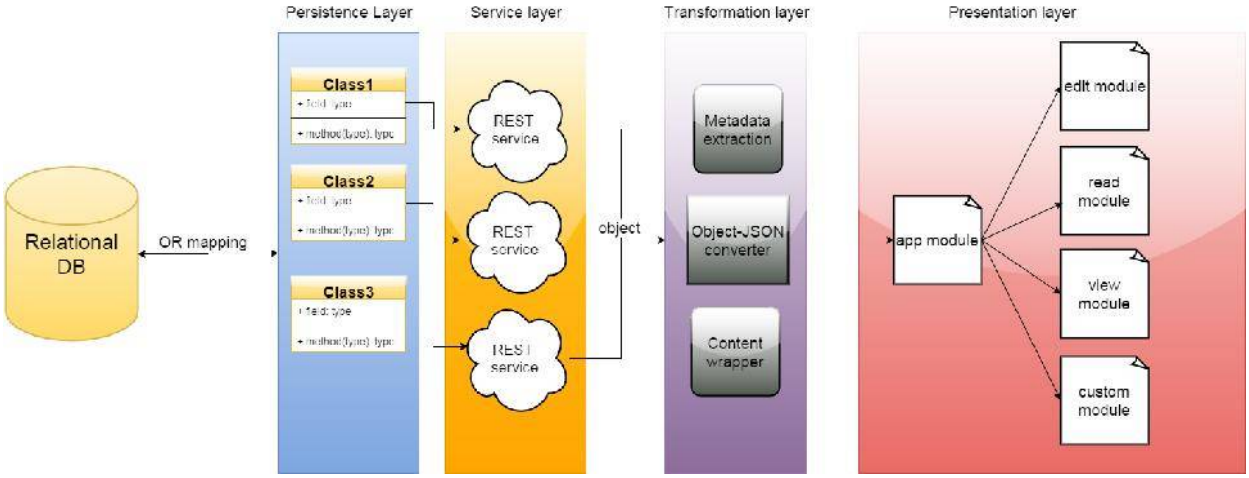

Figure 1. Adaptive framework architecture based on AngularJS

representation based on XML document, similar to the metadata object description used in our framework.

Adaptive user interface frameworks based on pre-built structures were proposed in [10, 14]. Aspects were used to implement adaptation of web application user interface based on runtime context.

Web frameworks are another related field of research. Some of the most prominent Web frameworks currently in use are Ruby on Rails, Django and Play framework. Our framework uses concepts similar to the mentioned frameworks, such as MVC architecture and built-in reusable components. The importance of such frameworks [2] is reflected by the need to quickly emerge on the market and adapt to changing user requirements, as well as their widespread use [15].

III. BACKEND

In order to present data from the backend, frontend must be provided with metadata, such as column names, column types, constraints, etc., which are contained in database schema. Since the backend has access to database schema, it is where mechanisms for fetching of metadata are contained.

By using Java Persistence API (JPA) [6], database schema can be defined through definition of persistence entities. Entities are Java classes that are mapped to a table in a relational database, with its fields representing database columns. These entities can be coded by hand or automatically generated from aforementioned persistence model.

Metadata necessary for object/relational transformation is provided by using annotations. This means that, instead of directly reading the database schema, it is possible to read class annotations (figure 3).

The backend contains RESTful web services. These services contain components needed to transform Java objects into JSON objects [8]. The reason behind choosing JSON as the serialization format is because the frontend is written in JavaScript, which has a native support for JSON objects. The web services in question are implemented using JAX-WS [7]. Each web service is represented with a java object. Service definitions are formed by using annotations.

Entity classes extend the *Model* class (figure 2). This class defines methods that obtain the metadata by

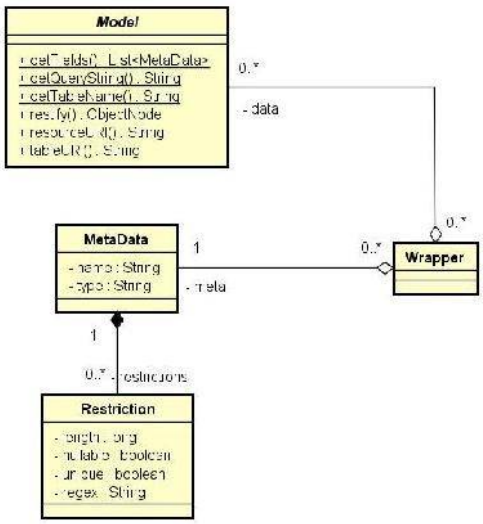

Figure 2. Transformation layer classes

reading JPA annotations of the entity. Once an object is ready to be serialized, its fields are being read during the runtime. Each field contains annotations with database metadata which is extracted and transformed into format suitable for JSON.

Classes that form the transformation layer are shown in figure 2. *Model* class contains path required for endpoint definition of the corresponding RESTful service.

Once formed, the metadata is contained in a *MetaData* object. Type name of primitive fields, such

```java
@Entity
@Table(name = "actor")
public class Actor extends Model {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "actor_id", unique = true, nullable = false)
    private long id;

    @Column(name = "actor_first_name", unique = true, nullable = false)
    private String firstName;

    @Column(name = "actor_last_name", unique = true, nullable = false)
    private String lastName;

    @JsonIgnore
    @JoinTable(name = "film_actor",
    joinColumns = @JoinColumn(name = "actor_id", referencedColumnName = "actor_id"))
    @ManyToMany(cascade = { ALL }, fetch = FetchType.EAGER)
    private Set<Film> films = new HashSet<Film>();
}
```

```json
{
   "data":[
      {
         "id":1,
         "firstName":"\tArnold",
         "lastName":"Schwarzenegger",
         "films":"/actor/1/films"
      }
   ],
   "meta":[
      {
         "name":"id",
         "type":"id",
         "restriction":{
            "length":255,
            "nullable":false,
            "unique":true,
            "regex":null
         }
      },
      ...
```

Figure 3. Transformation of a model class

as integers and strings, are taken as they are Java. Type of the primary key is always *id*. Associations are treated depending on their multiplicity. One-to-many and many-to-many relations are represented with a collection, while many-to-one is a reference to an object. Collections are called *zoom* fields, while references are called *link* fields, coming from the way of their representation [14]. Following the HATEOAS [19] principles, metadata for such fields should contain URI of the associated object.

Each *MetaData* object contains a collection of *Restriction* object, each containing a set of database restrictions for a single field. This includes all the restrictions supported by JPA annotations, such as *nullable*, *unique*, and *length*, with additional restriction *regex*, needed to define derived types. Once a *MetaData* object is fully formed, it is added to a JSON object with key "meta"; the java object itself is added with key "data" (figure 3). Finally, each *MetaData* object includes URI of the data object based on its type and its primary key. *Wrapper* class is responsible for generating the required JSON object.

IV. FRONTEND

Unlike other solutions [12] that use JSP pages for the frontend, our solution makes use of AngularJS framework. AngularJS enables creation of a single page applications and allows some of the logic (such as validation) to be included on the client side.

The client side application consists of modules (figure 1):

- **app** - routing and definition of modules.
- **view -** detailed view of an object.
- **edit -** editing and creation of objects.
- **read** - listing of all objects.
- **collection/zoom** – enables collection manipulation, such as adding or removing objects from a collection.

A module consists of a controller and multiple views and is responsible for one type of operation. Multiple views for a single controller enable the customization of the user interface, depending on object type. Each view is available on its URL path. This path is used to bind object types to their presentation page. URL paths are defined in *app* module as *routes* formed of controller-view pairs. Controllers and views can be combined to meet a certain requirement (for example, customization of the edit page). Routes for basic CRUD operations are initially generated. New routes can be added by writing a new AngularJS module that extends the *app* module. By adding new routes it is possible to further expand the client application beyond basic functionalities.

The *app* module is responsible for routing of both requests and responses. Backend services are mapped to controllers-view pairs in the module.

AngularJS offers mechanisms for extending HTML with directives. A directive acts like any other HTML tag. Upon execution, AngularJS scans the page and invokes the corresponding controller whenever it encounters a directive. The controller then modifies the HTML page by replacing said directive with new code.

Module controller is responsible for preparing data for presentation. Once an object is received from the backend, it is added to controller scope along with its metadata. This way, it is accessible from the module view. Module controller also contains functionalities for filtering, sorting, and validation. If a controller uses a backend service, its route should be equivalent to the service route. Basic controllers can be extended by using inheritance.

Table 1: Web services and operations for video library example

Operation Route	**HTTP method**	**Count**
film	POST GET DELETE	3
actor	POST GET DELETE	3
category	POST GET DELETE	3
id/film/actors	POST GET DELETE	3
id/film/categories	POST GET DELETE	3
id/actor/films	POST GET DELETE	3
id/category/films	POST GET DELETE	3
id/film	POST PUT GET DELETE	4
id/actor	POST PUT GET DELETE	4
id/category	POST PUT GET DELETE	4
Total: 3 services, 33 operations		

Table 2: Routing table for video library example

Paths	**Controller**	**View**	**Total**
/actor, /category, /film	Read	read-edit	3
/actor, /category, /film	Add	add	3
/id/film, /id/actor, /id/category	Edit	edit	3
/id/film/actors, /id/films/categories, /id/category/actor,	Collection	read	3
/id/film/actors/new, /id/films/categories/new, /id/category/actor/new,	Read	zoom	3
/	Main	main	1
Total: 16 routes			

Module view consists of an HTML page, directives and its controllers. AngularJS allows writing of custom directives. One such directive is used for presentation of an object. This directive is the central piece in user interface adaptiveness. The idea is to be agnostic to the received object type. This directive brings two options: the first allows the presentation of all objects fields in one place; the second allows

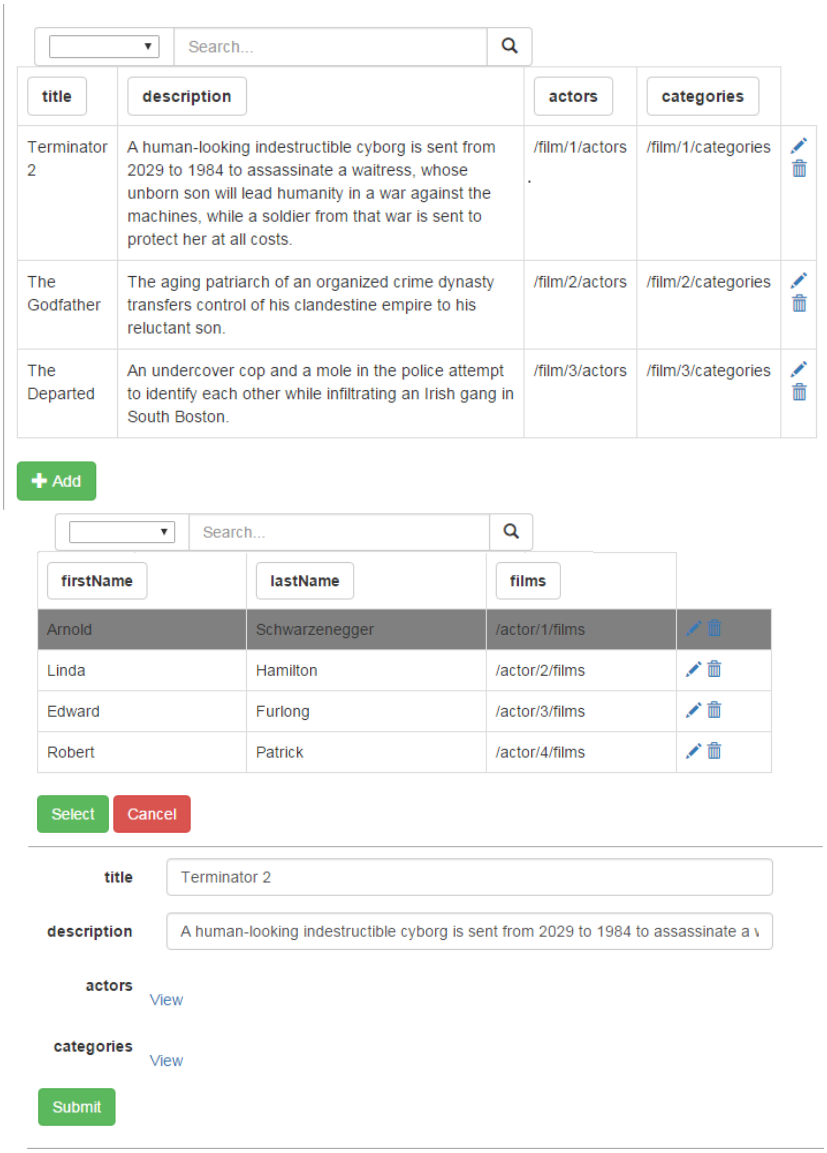

Figure 5. Application screens, top to bottom: table view, zoom picker and edit/create form

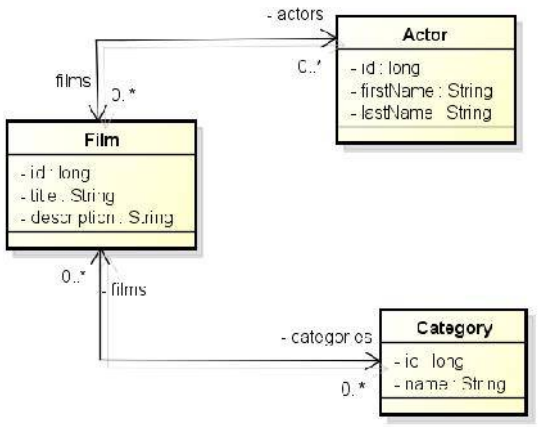

Figure 4. A part of a video library persistent model

custom layouting of the user interface, by passing a field as a parameter. Directive controller works similarly to a template engine. It generates an HTML component based on passed parameters and puts it in place of the directive. Type-component mapping is shown in figure 5.

V. An Example

The framework presented in this paper was used to build a simple web application based on model in figure 4. This web application provides only CRUD operations to a client. Entities are implemented by generating annotated Java classes, as shown in figure 3. These classes extend Model class, and are called *model classes*. Services are implemented by

generating JAX-WS REST services. Each service contains CRUD operations for model classes. Services communicate with clients via wrappers, which encapsulate request/response data. The client application contains routes. Screen form examples are shown in figure 5.

The web application in question is a Java enterprise application. Developing this web application required generating model classes. Each model class required a corresponding DAO and REST service that provide CRUD operations for the client application. REST services were generated as Java classes, using JAX-WS annotations. Summary is shown in table 1.

Each operation required a route in the client application *app* module. Routes in this module are formed by using ngRoute, which is an AngularJS module. These routes map relative paths to controller/view pairs. Each pair allows using a service operation with the same relative pair, enabling data manipulation or presentation, depending on the assigned controller/view combination.

Basic controllers and views are provided by the framework. These include Add, Edit, Delete, Update, Read, Collection and Zoom controllers; and Edit/Create, Read, Read-Edit, Zoom and Table views. Routes were formed by combining existing controllers and views (table 2). Example routes are shown in listing 1. Expanding the functionalities of an application beyond basic CRUD operations is done by creating new routes. Writing additional routes takes minimal effort, since their form is rather simple. Client application required no additional modifications. Screens from the client application are shown in figure 5.

```
.config(function ($routeProvider) {
  $routeProvider
  .when('/', {
    templateUrl: 'views/main.html',
    controller: 'MainCtrl'
  })
  .when('/:id/film', {
    templateUrl: 'views/edit.html',
    controller: 'EditCtrl'
  })
  .when('/film', {
    templateUrl: 'views/read-
edit.html',
    controller: 'ReadCtrl'
  })
  .when('/:id/film/actors', {
    templateUrl: 'views/read-
edit.html',
    controller: 'CollectionCtrl'
  })
```

Listing 1 A route example

VI. CONCLUSIONS

The paper presented a framework for development of single-page web applications. The framework enables frontend development through the use of adaptive user interface. The presented realization is based on Java technologies for the backend.

The backend generated in this solution is based on Java technologies, although ideas, mechanisms and methods presented by this paper can be easily realized in any other web-based technology.

The goal was to shift focus from frontend development. This way it was possible to develop models and backend services with minimal effort required to build the user interface.

Instead of generating the user interface or manually writing view for each data type, this approach uses built-in components capable of presenting any data at the backend. The components are generic enough to provide basic business operations on provided data.

Once the model is done and RESTful services are implemented, the application is ready for deployment; the user interface offers basic functionalities as soon as the application is deployed. The price for quick startup is limited customization options. The framework does not offer much customization options beyond layouting of the user interface. Additional styles can be loaded from an external css file, but currently there is no mechanism for defining a class for HTML inputs offered by the framework.

Further development can be directed in a way to remove or reduce framework's deficiencies. Potential improvements include: (1) possibility to write new controllers from scratch, (2) more initial controllers and controller-view combinations, (3) HTML class attributes for directives, (4) improvements in layer for object-JSON transformation, (5) ability to define custom constraints and data types etc.

Based on the example in section 5, we concluded that the client application developed using this framework required no modification. This meant that the client application was ready for deployment as soon as the backend was operational.

Model classes, services and basic routes were generated from the persistence model. This greatly reduced the amount of work needed to create the backend and minimized the time needed for project startup. The final goal is to create a full-stack Web development framework, with seamless integration of other frameworks used for its implementation.

VII. REFERENCES

[1] G. Milosavljevic, M. Filipovic, V. Marsenic, D. Pejakovic, I. Dejanovic, Kroki: A mockup-based tool for participatory development of business applications.. SoMeT (p./pp. 235-242), : IEEE. ISBN: 978-1-4799-0419-8, 2013.

[2] Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan, Frameworks: Why They Are Important and How to Apply Them Effectively: Department of Electrical Engineering and Computer Science Vanderbilt University Nashville, TN 37203, ACM Queue magazine, 2004

[3] Django framework. https://docs.djangoproject.com, retrieved 14.1. 2016.
[4] Ruby on Rails framework, http://rubyonrails.org/documentation/, retrieved 14.1.2016.
[5] Play framework, https://www.playframework.com, retrieved 14.1.2016.
[6] Java Persistence API, http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html, retrieved 14.1.2016.
[7] JAX-WS, https://jax-ws.java.net/, retrieved 14.1.2016.
[8] JSON, http://www.json.org/, retrieved 21.1.2016
[9] B Milosavljevic, M. Vidakovic, S.Komazec, G. Milosavljevic, User interface code generation for EJB-based data models using intermediate form representations,2nd International Symposium on Principles and Practice of Programming in Java, PPPJ 2003, Kilkenny City, Ireland, 2003
[10] T. Cerny, K. Cemus, M. J. Donahoo, and E. Song. Aspect-driven, Data-reflective and context-aware user interfaces design. Applied Computing Review, 13(4):53–65, 2013
[11] G. Milosavljevic, M. Filipovic, V. Marsenic, D. Pejakovic, I.Dejanovic, Kroki: A mockup-based tool for participatory development of business applications.. SoMeT (p./pp. 235-242), : IEEE. ISBN: 978-1-4799-0419-8, 2013.
[12] Kroki, www.kroki-mde.net
[13] Kroki demo, http://youtu.be/r2eQrl11bzA
[14] M. Filipović, S. Kaplar, R. Vaderna, Ž. Ivković, G. Milosavljević, I. Dejanović, Aspect-Oriented Engines for Kroki Models Execution, Intelligent Software Methodologies, Tools and Techniques (SoMeT), 2013
[15] http://trends.builtwith.com/framework, retreived 30.1.2016.