Experiment Management System for the SND Detector

K. Pugachev\textsuperscript{1,2}

\textsuperscript{1} Budker Institute of Nuclear Physics, SB RAS, Novosibirsk, 630090, Russia
\textsuperscript{2} Novosibirsk State University, Novosibirsk, 630090, Russia

E-mail: K.V.Pugachev@inp.nsk.su

Abstract. We present a new experiment management system for the SND detector at the VEPP-2000 collider (Novosibirsk). An important part to report about is access to experimental databases (configuration, conditions and metadata). The system is designed in client-server architecture. User interaction comes true using web-interface. The server side includes several logical layers: user interface templates; template variables description and initialization; implementation details. The templates are meant to involve as less IT knowledge as possible. Experiment configuration, conditions and metadata are stored in a database. To implement the server side Node.js, a modern JavaScript framework, has been chosen. A new template engine having an interesting feature is designed. A part of the system is put into production. It includes templates dealing with showing and editing first level trigger configuration and equipment configuration and also showing experiment metadata and experiment conditions data index.

1. Introduction

There is the SND detector\textsuperscript{1, 2, 3} (figure 1) at the VEPP-2000 collider\textsuperscript{4}. The upper bound of the raw data flow was estimated as 25Gb/s\textsuperscript{5}. This results in a few hundred gigabytes of the data stored per day. The experiment requires metadata (dozens megabytes per day) for reconstruction and processing including configuration of about twenty SND subsystems.
Every subsystem has a specific person (expert) assigned. Experts configure subsystems. The experiment collaboration is relatively small consisting of about 30 scientists and engineers.

Detector management software consists of a lot of application programs, scripts and servers. In this article we would like to pay attention to a management system implemented as a web-interface for the SND operators. There is one at the moment. But some of the system functions have not been implemented yet; some solutions are too old or contain old bugs.

The new system is meant to be web-oriented. It should replace the most inconvenient UI solutions, implement the new features being waited and fix some bugs.

Experiment configuration is represented hierarchically being stored in 78 database tables. To implement an interface for editing this structure is an important purpose of the new system. The current system requires user’s knowing relational database systems and/or C++ while editing the configuration.

2. The System Implemented and Tools Used
A HTTP(S) server has been created using Node.js [6]. Currently it allows user to access and edit several SND databases. The following features are implemented:

- showing and editing first level trigger configuration,
- showing and editing equipment configuration,
- showing experiment metadata,
- showing experiment conditions data index,
- showing some integral experiment runs parameters.

The server depends on mysql, node-static, pegjs, sessions, formidable packages that could be found at NPM [7] and on crypt.js [8].

3. Layers of Abstraction
The SND team members have different IT skills and different expertise in physics and SND subsystems. There could be distinguished at least the following groups:

- operators (may have no IT skills and minimal knowing SND subsystems),
- experts (know SND subsystems well),
- programmers (have IT skills).

It is useful to separate concerns of the groups mentioned as far as possible. This separation allows the members to interact with the system using the most appropriate terms and involve as few people as possible when solving problems.

The server deals with four logical layers which differ in IT-awareness of the SND team member and the terms/representations used:

- user interface (web-interface, GUI; for operators and experts),
- user interface templates (templates, template language; for experts),
- template variables/functions description and initialization (specialized JavaScript; for experts and programmers),
- implementation details like database interaction (JavaScript; for programmers).

The first layer is just a web interface which could be used by any SND team member or operator. It could be treated as a virtual one separated to introduce a user role. The next layers are quite real, represented by the server architecture and its folder/file structure.

User interface bases on a set of templates written in a simple template engine language. Templates use predefined values that are exported JavaScript variables and functions. Exported JavaScript stuff conforms the rules of the server core which implements some details and acts like a framework.
4. Template Engine

4.1. Motivation

To support separation of concerns it is convenient to use template engine that has simple enough syntax, asynchronously computed values, hides any difference between synchronous and asynchronous calls and makes user separate templates and logic.

There are a lot of template engines created. Ones propose embedding JavaScript code in HTML templates (e.g. EJS [9], AJS [10]), others introduce special syntax (e.g. Pug [11], Kernel [12], dust.js [13]), some engines just perform HTML transformations keeping HTML and server-side JavaScript separated without using additional languages (e.g. Plates [14]).

Dealing with databases involves chains of asynchronous calls. Some existing template engines are synchronous (one should prepare all the values required and pass them to e.g. EJS, Pug). Some of them do support asynchronous computations but tend to use function-like terms (calls/blocks/filters) instead of operating values computed asynchronously (every table row is such a value).

Such powerful engines like EJS, AJS and Plates provide too much freedom and flexibility to be used by non-professional programmers.

It was decided to create another template engine.

4.2. Syntax and Description

The interface template layer introduces web pages templates. Each of ones describes a page or an important block (e.g. site menu, SND parameters table, etc) in declarative way. The layer requires understanding a simple template syntax (including values, function applications and block applications) and knowing some high level concepts like predefined blocks names. It is about page content layout. So concerned physicist can adjust some templates without being a programmer.

A simple template code looks as follows:

```
import my_scope
include header.tpl as header
--------
||header||
<h1>Hello, |name|!</h1>
|#if (equals counter 1000000) |
  <p>|toupper name|, you are the |counter|th!</p>
|#else |
  <a href=".">refresh to become the |inc counter|th.</a>
|#end |
||footer||
```

It starts with importing scopes and including standalone templates. This is followed by the template body. When imported a scope, user can insert values/functions defined in that scope.

Value insertion looks like |x1|, function application is |f x1|. There are blocks, e.g. |#if x1|...|else|...|end| or |#foreach xs: x|...|end|. The template engine provide means of exporting constants, defining lazily evaluated values, functions and custom code blocks. Lazily evaluated values could be variable (e.g. one can define |rand| for random number) or unique per template rendering. Functions could be declared as pure or impure ones. It may be used for templates simplification. Lazily evaluated values and function could be synchronous or asynchronous as well. One can define them as ordinary JavaScript functions or ones getting template engine context and arguments representation for precise control (e.g. non-strict evaluation or handling engine-specific errors).

Templates are stored in separate files. The files have a special (but configured) extension to help the server find and load them automatically.
The template entities mentioned form the third logical layer dealing with template variables/functions description and initialization. The server core provides a sort of framework making the JavaScript code at the layer “specialized”. It describes domain specific features and group them into scopes to import.

One should create a file that implements a CommonJS module exporting a function. The function has the only argument (let’s call it “_”). When starting the server loads the file (it also has a special extension, e.g. “.tpl.js”) automatically and calls the function passing the framework helpers as the argument. So one can use such helpers as \_\_scope to unite some values into namespace-like scope, \_\_func to define a function, \_\_lazy or \_\_sql to define a lazily evaluated value, the latter resulted from SQL query. There are useful defaults for constants and simple functions export.

Let’s consider an example of a “third layer” file:

```javascript
module.exports = function (_) {
  _.scope('test', ['global'], {
    sum: function(a, b){ return a+b; },
    sum_a: _.func(function(a, b, cb) {
      setTimeout(function(){ cb(a+b); }, 100);
    }, 'pure/async'),
    one: 1,
    one_a: _.lazy(function(cb) { cb(1); }, 'pure/async')
  });
};
```

The example above defines scope test that defines two functions sum, sum_a and two values one and one_a. Two of these values evaluate in a synchronous way (sum and one) and others are in asynchronous one (sum_a and one_a). test scope inherits some missing functions and values from global scope as well.

JavaScript proposes non-blocking computations in a single thread. So asynchronous evaluation is widely used in JavaScript code. This approach separates user from multithreading problems but complicates the code with callbacks (that are carry information about what way the value is being got and nothing useful). In the server in question database queries is the main asynchronous thing.

4.3. Homogeneity of Values

The main feature of template engine created is homogeneity of the entities in terms of evaluation way. So one can mix pure functions with impure ones and – this is the point – synchronous functions/values with asynchronous ones. So let’s import the defined above test scope and look what it yields:

```javascript
import test
-------------
| sum one one |
| sum one one_a|
| sum one_a one_a|
| sum_a one one |
| sum_a one one_a|
| sum_a one_a one_a|
```

The server will load the CommonJS module presented in the previous subsection and the template shown here. The first one provides scope test and the second one uses entities the scope contains.

All the function applications will be substituted by “2”. So at the second – template – layer any function is just a function and any value is just a value, no matter what way it has been evaluated. All the expressions looks like mathematical ones except for a bit unusual syntax.
“Pure” directives may result in simplification or caching and asynchronous computations may execute simultaneously.

One important point to know is undetermined order of computations. So one **must not** rely on the order of values/functions are listed in the template and create imperative style templates. The only factor that influence on the order is dependency between the values. Dependent expressions are evaluated after those they depend on. However the user may specify the arguments evaluation order while defining a function or a block in a special way.

5. Editing Hierarchical objects

5.1. Configuration Editing Concern

SND uses relational databases (managed by MySQL) to store the experiment configuration. The configuration satisfies a hierarchical model that is represented as a set of database entries being stored in several tables and having cross-references. Every model tree node is represented as a table row.

Once being used in experiment, the configuration should stay unchanged. So editing some parameters implies acquiring a copy of the configuration in question. To perform this one is to execute several SQL queries. The current SND software does not provide more convenient ways of configuration editing than query constructing. It requires user's attention and knowing SQL.

The system under consideration provides more convenient interface. It looks **information_schema** for tables references and uses this information to build a JavaScript representation of a configuration model. Having this model it could recursively retrieve concrete database rows values for showing or build a hierarchical interface for editing.

This approach requires specifying some metadata describing bounds of the model (the root node; nodes to include – because some nodes may have e.g. **author** field refering to user entry that is not a part of configuration but just a piece of configuration metadata; back-referenced nodes to include), pretty tables/fields names to display in the web-interface. When the metadata is specified the user interface for showing and editing such hierarchical structures could being generated automatically.

A retrieving/editing module and its web interface have been created. The server allows user to specify values to change and/or select existing subnodes to replace ones in the configuration being edited. All the values changed are highlighted and could be restored by clicking a button. Unchanged values substituting, author name appending and queries generating are performed automatically. Every changed value enforce copying its node and parents nodes up to the root node. When a node and its children have no changed fields, it is not copied but just referenced to avoid creating unnecessary entities.

5.2. Template Helper

The server core includes a retrieving/editing module that provides a “third layer” helper (**_.editor** for defining template engine values. **_.editor** produces a set of variables/functions required: **edit** for database editing, **get_node** for retrieving database row corresponding a configuration node, etc.

Here is an example of using **_.editor**:

```javascript
var editor = _.editor({
  db: 'rundbase',
  table: 't_trigger',
  leafs: [
    't_trigger/t_triggeremcsimths',
    't_triggeremcl/t_triggeremclmasks'
  ]
}, [

```
path: 't_trigger',
data: { title: 'Trigger' },
fields: [
    { name: 'c_comment', data: { title: 'Comment' } },
    { name: 'c_brptmask', data: { title: 'BRPT mask' } }
],
{
    path: 'c_flt10: t_triggerargs/admdbase.t_person',
data: { title: 'Author' },
fields: [
    { name: 'c_rlastname', data: { title: 'Last name', flag5: true } }
]
});

A pair of db and table specifies the root node. leafs are back-referenced nodes. The second argument of _.editor specifies description for each node. It contains the path from the root to the node, arbitrary node data and the list of its fields including their arbitrary data. By the time the data is mostly the node/field title, but it could be, for example, some directives for the web interface.

Using the helper described first level trigger configuration (see the left side of the figure 2) and equipment configuration editors templates were built.

6. Appearance

![Figure 2](image)

Figure 2. Appearance of the system in question. Left to right, up to down: editing hierarchical first level trigger configuration; showing the configuration; showing some integral experiment parameters regarding the current operator’s shift.

The server has an interface based on simple block layout. Currently it results from the functional requirements of the system. The interface tends to use CSS and pure JavaScript when it is possible. Some pages make use of jQuery [15] and jQuery UI.

One may take a look at the interface at the figure 2.

7. Conclusion

In order to provide more convenient UI solutions, get rid of some bugs and implement features being waited a new management system is being developed. At the time a layered abstraction
system is designed and a simple web-server has been created. It uses a simple template engine designed (the main feature is homogeneity of synchronous and asynchronous values). The server allows user to read and modify experiment configuration, conditions and metadata. A hierarchical objects editor has been implemented. The editor had not been implemented before in web-interfaces of SND.

This is the first stage of development. Some new changes are coming soon but the current system is partially put into production.

Acknowledgments
The author would like to thank a number of the SND team members for help, guiding, discovering bugs and inspiration. This work is partly supported by the RFBR grants 16-02-00327 and 16-32-00542.

References
[1] Achasov M N et al 2000 Spherical neutral detector for VEPP-2M collider Nucl. Instrum. Methods A449 125–139 http://dx.doi.org/10.1016/S0168-9002(99)01302-9 (arXiv:hep-ex/9909015)
[2] Abramov G N et al 2001 SND upgrade eConf C010430 T10 arXiv:hep-ex/0105093
[3] Aulchenko V M et al 2009 SND tracking system: tests with cosmic muons Nucl. Instrum. Methods A598 102-104 http://dx.doi.org/10.1016/j.nima.2008.08.099
[4] Khazin B I 2010 Detectors and physics at VEPP-2000 Nucl. Instrum. Methods A623 353-355 http://dx.doi.org/10.1016/j.nima.2010.02.246
[5] Bogdanchikov A G et al 2014 SND data acquisition system upgrade JINST 9 C06013 http://stacks.iop.org/1748-0221/9/i=06/a=C06013
[6] Node.js: https://nodejs.org/en/.
[7] npm: https://www.npmjs.com/.
[8] tripcode/crypt.js at master - KenanY/tripcode: https://github.com/KenanY/tripcode/blob/master/lib/crypt.js.
[9] Embedded JavaScript: http://www.embeddedjs.com/.
[10] Asynchronous templating in Node.js: https://github.com/IonicaBizau/ajs.
[11] Pug: https://pugjs.org/api/getting-started.html.
[12] kernel. A simple async template language similar to dustjs and mustache https://github.com/langpavel/kernel.
[13] Dust.js. Asynchronous templates for the browser and node.js https://github.com/akdubya/dustjs.
[14] Plates. Light-weight, logic-less, DSL-free, templates for all javascript environments! https://github.com/flatiron/plates
[15] jQuery: http://jquery.com/.