Daniel Amariei, FII, UAIC
daniel.g.amariei@gmail.com

Web Application Development
Sabin-Corneliu Buraga, FII, UAIC
http://profs.info.uaic.ro/~busaco/teach/courses/wade/

RESTing on MVC using Node.js and Express

1. Introduction

When we talk about software construction, an important aspect that we should take into account is that of the software architecture. In rather generic terms, we can think of a software architecture as a framework that offers us the structure in which to operate when dealing with complex problems.

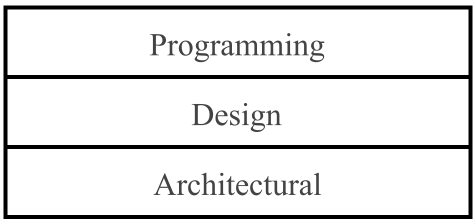
When talking about architecture and software construction, Patterns can be of great help, since they distill expert knowledge – gained in years – into a few simple guiding rules.

The document is structured as follows: *section 2* presents the concept of Pattern, including the different types of available; in *section 3* we describe the main components of the *MVC* Architectural Pattern, together with its benefits; following, in *section 4* we describe the *REST* architectural style, tackling the six constraints that we must conform to; going further to *section 5*, we briefly describe the main characteristics of the *Node.js* framework; in *section 6* we will discuss the process of building a *REST*ful API for an online agenda; in *section 7* we conclude.

2. Patterns

The most simple definition of a pattern might be *A solution to a problem in a context* [1]. They allow us to decompose complex structures into simple building blocks, while providing general (abstract) solutions to classes of problems.

This decomposition happens at different levels, and as such we have different types of patterns: architectural, design and programming.



Architectural Patterns
Architectural patterns are high-level strategies that enables us to decompose the system by means of terms and concepts from the application domain. They take into consideration the global structure of the systems, establishing its emergent properties.

Design Patterns
Design Patterns provide methods that enable the refining and building of small subsystems by means of software design constructs that involve the usage of: objects, classes, inheritance, aggregation and others similar to these.

Programming Patterns (Idioms)
Programming Patterns are paradigm and language specific constructs. One perspective from which one can regard them is as best practices for a specific language.

In conclusion, when we tackle the task of building software, we have at our disposal several levels where we can apply patterns, from the most general, to the most specific: the architectural level, the design level and the programming level.

For more details we encourage the reader to follow [1, 2, 3].

3. The MVC (Model View Controller) Architectural Pattern

MVC (Model View Controller) is pattern that enables us to structure an application from an architectural point of view, as such it is an architectural pattern.

As its name implies, the patterns divides the application into three separate subsystems, effectively providing separation of concerns. The three subsystems in which the system is divided are:

1. Model;
2. View;
3. Controller.

The purpose of the pattern is to separate the data (**Model**), from the manner in which the information is presented to the user (**View**) and the way in which the end-user interacts with and manipulates it (**Controller**).

Organizing the system in this manner helps produce flexible and reusable modules which are cohesive and low-coupled. Also, the designer will have a clear idea about how the responsibilities are shared among the subsystems.

Following, we will briefly describe the role of each subsystem.

The Model
The purpose of the model is to encapsulate the data in our business domain. The model can be updated following in response to user interactions, or other events that happen in the system. After such an update is made, the **Model** is responsible with notifying the **View** of this.

The View
The **View** employs User Interface (**UI**) components in order to appropriately display the model. While the system may be used by several (possible different) contexts, we can create a **View** for each possible context. It is the responsibility of each view to accurately present the state of the model, as such, when it receives an update notification it will accurately update its representation of the model.

The Controller
As with the **View**, the **Controller** also employs **UI** components in order to capture user input. As a result of acting in response to the user input, the **Controller** may issue update requests to the **Model**, to the **View**, or both. In essence this is the component that contains the business logic of our application.

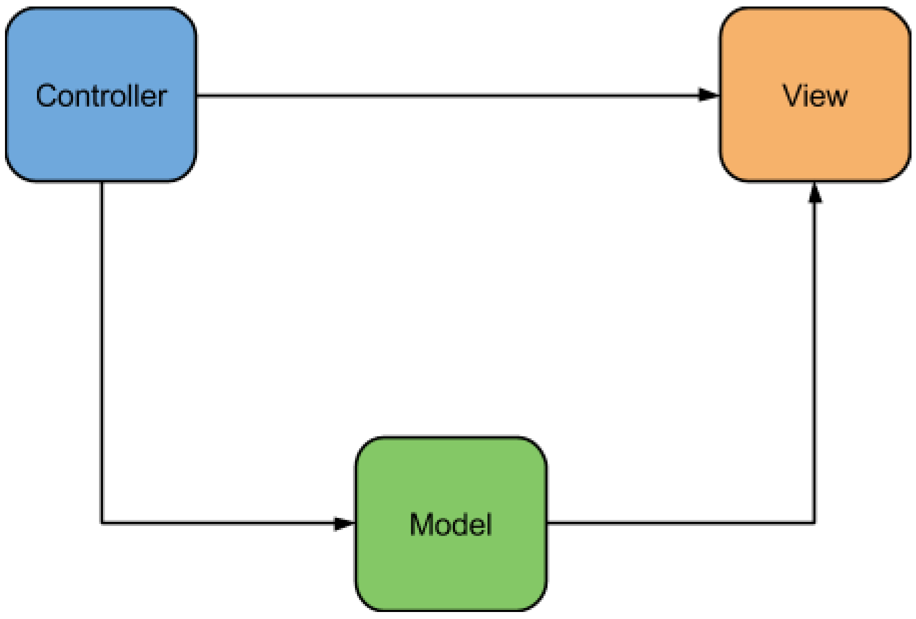


Figure 2.1: The Model View Controller architecture.

One important note that we need to make is the fact the the **UI** needs to be distinguished from the rest of the systems and as such is should not be confused with the **View** – they are distinct.

4. REST

REpresentational **S**tate **T**ransfer (**REST**) is an architectural style proposed by Roy Fielding in his Doctoral Thesis. In contrast to *SOAP*, in a **REST**ful *API* we talk about *resources* rather than *actions*, where resources can be assimilated to nouns and actions to verbs.

One must take into account the fact that the **REST** architectural style is not restricted to *HTTP*, but using this protocol comes with its advantages. Using the *HTTP* protocol, the resources can be identified by *URIs*, with the possibility to have multiple *URIs* pointing to the same resource.

When talking about resources, we must make a clear distinction between resources, which are abstract in nature, and their representation(s) – a resource can have multiple representations. The representation of a resource is not the resource itself. The representations of a resource offer:

1. the possibility to manipulate the resource;
2. a specification of a resource's state.

In order to conform to the **REST** architectural style, we must take into account the following six constraints:

1. *Uniform Interface*
   The role of this constraint is to specify that between clients and servers an uniform interface must be created. The main purpose of this constraint is to produce an architecture with decoupled components, a characteristic that will enable them to evolve independently, as needed.

   In order to achieve this constraint, the following four guiding principles can be of help:

   I. Resource-Based
   The resources can be identified using *URIs*. Since resources are abstract in nature, the client will receive back only a representation of the resource.

   II. Manipulation of Resources Through Representations
   The representation of a resources, including its metadata, shall store all the information required to manipulate that resource.

   III. Self-descriptive Messages
   Each message includes all the information required by the client to process it.

   IV. Hypermedia as The Engine of Application State (HATEOAS)
   The clients and servers shall deliver state to each other via headers, body contents, query-string parameters, status codes, and the like. Moreover, whenever necessary, links should be contained in the response in order to engage in a retrieval of the actual object or similar ones.

2. *Stateless*
   As the name of the constraint implies, the host of the *REST*ful (web) service shall not contain client state. This means that each request is self-descriptive, containing all the necessary state to handle a request; this can be done using various mechanisms like: using query-string parameters, passing the state in the body, using headers, etc.

   Taking all of this into account means that the client is responsible for storing its state. Here we need to make the distinction between *application state* and *resource state. Application state* is that data that could vary by client and request, while *resource state*, is constant across every client who requests it.

   Making the effort to avoid spanning the *application state* over multiple requests will enable our service to be scalable in nature.

3. *Cacheable*
   Resources must define themselves as being cacheable, case in which some of the interaction between the client and the server is eliminated, further improving scalability, or not, case in which the client is prevented from using stale information.

4. *Client and Server*
   The *REST* architecture has two main entities: the *client* and the server. The *uniform interface* separates the client from the server, enabling them to be replaced and be developed independently. This clearly conforms to the *separation of concerns* principle.

5. *Layered System*
   A *REST*ful architecture can be constructed from multiple layers of software. Nevertheless, a client shall not be able to tell if he is connected directly to the server, or some intermediate *layer*.

6. *Code on Demand (optional)*
   Sometimes, the servers have the ability to extend the functionality of their clients, transferring executable logic to them.

This section was based on [8]; as such, we encourage the reader to follow it for more details.

5. Node.js

Node.js is a runtime environment which internally uses the *Google V8 JavaScript engine* to execute code. It enables the development of server-side applications using JavaScript. Put it more simply, **Node.js** is a context that allows you to run *JavaScript* code on the server-side.

Conceptually, all the code code in **Node.js** is running in a single process, therefore there is no parallel code execution. This is at the conceptual level because in the back-end – invisible to the programmer – the system could use thread pool and/or multiple processes to do the work.

Moreover, it provides an event-driven architecture with the following characteristics:

1. Non-blocking I/O: in order to make the program as responsive as possible, every I/O call must take a callback, be it a disk, network, or any other operation;
2. POSIX compliant;
3. Built-in support for: several web/internet protocols, operating systems resource management, handling operating systems abstractions, and many more;
4. Data (*e.g.,* from network, from disk, etc.) is accessed implicitly through streams.

Code execution happens through a mechanism generically termed *event-loop*. The *event-loop* allows you to specify what happens when a particular event occurs. This functionality is achieved through the use of callback functions.

6. Online Agenda

The purpose of this section is to detail a concrete example which applies some of the information presented in the preceding sections. As such, we will build a RESTful API for an online agenda, using *Node.js* and *Express*. In the first part of this section we will detail the process of installing the required software, while the second part is reserved for detailing the process of developing the application.

6.1 Installing the required software

Before we proceed to explain how to build the application, in this section we will detail the steps that need to be performed in order to configure the system with the required software for developing and running the application.

Step 1

The first step involves installing the database support for our application. In order to store the data for the agenda, we will use *MongoDB*, which can be installed from: *http://docs.mongodb.org*.

Step 2

Since in *section 6* we have briefly described the *Node.js* platform, we will install it in order to use it as a runtime environment for our application; in order to install *Node.js* please follow the details found at: *http://nodejs.org/*.

Step 3

The next step involves installing the Express application generator using the following command: *$ npm install express-generator -g*.

npm is the *Package Manager* for the *Node.js* platform, and it is available right after installing *Node.js*.

Step 4

After executing the preceding command, we can use the *express-generator* in order to create the structure for our application. The synopsis of the command is: *$ express myapp*. This will create the folder *myapp* which will contain the default project structure for all *Express* projects, including a *package.json* file. The file stores various configurations for the application and most importantly its *dependencies*. The dependencies will also include the *Express* framework. Here we need to emphasize the distinction between the *express-generator* and the *Express* framework.

Step 5
Since after executing the previous command we ended up with a project structure that contained a file with the dependencies of the project, all we need to do is to use a command that will install them automatically: this command is: *npm install*.

Step 6
Since working directly with *MongoDB* is a bit tedious, we will use an object modelling layer that will handle all the validation, casting, and boilerplate code involved when using such a storage engine. The intermediate layer is *Mongoose* which can be installed by issuing the following command: *$ npm install mongoose --save*

Step 7
While *express-generator* has created all the project structure and required files for a skeleton application, it does not add code to start the server. As such, this step involves adding the server startup code to the *app.js* file, located in the root directory.

Server startup code:

```
// connect to the mongodb daemon
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/agenda');

// start the server
var server = app.listen(3000, function () {

        var host = server.address().address
        var port = server.address().port

        console.log('Example app listening at http://%s:%s', host, port)
});
```

6.2 Developing the application
In this section we will briefly describe the steps that need to be performed in order to create the application. This must be seen as complementary to the accompanying code and presentation.

Step 1
In this step we create the *Schema* and *Model* for the *Contact* entity in our application. The code for this step can be found in *agenda/models/contact.js*.

Step 2

The next step involves adding the required middleware that handles access to the resources defined by the application (*i.e.*, Agenda and Contact). The code for them can be found at: *agenda/routes/agenda.js* and *agenda/routes/contact.js*.

7. Conclusions

When developing Web applications we must put considerable effort when creating the architecture. As such, the type of architecture that we choose can lead to a successful or unsuccessful product. Patterns can be of great deal when developing complex applications. As such, the document briefly reviewed the role of Patterns and the type of available ones. Following, we described the main components of the *MVC* Architectural Pattern, together with its benefits. The next section presented the *REST* architectural style, together with the constraints that we must conform to. Finally, we presented the main characteristics of a popular framework (*i.e.*, Node.js), concluding with an use-case.

A. Bibliography and references

1. ***, Bob Tarr, Design Patterns in Java. *Introduction To Design Patterns.* http://userpages.umbc.edu/~tarr/dp/lectures/IntroToDP.pdf.

2. ***, Bob Tarr, CMSC446. *Introduction To Design Patterns*. http://userpages.umbc.edu/~tarr/dp/spr06/cs446.htm.

3. ***, Stack Overflow. *Difference between design pattern and Architecture?* https://stackoverflow.com/questions/4243187/difference-between-design-pattern-and-architecture.

4. Brahma Dathan, Sarnath Ramnath. *Object-Oriented Analysis and Design.* Springer, 2011.

5. ***, Sabin Corneliu Buraga, Web App Development. *Node.js*. http://profs.info.uaic.ro/~busaco/teach/courses/web/presentations/web-nodejs.pdf.

6. ***, Wikipedia. *Node.js*. https://en.wikipedia.org/wiki/Node.js.

7. *** , Books at mixu.net. *Mixu's Node book*. http://book.mixu.net/node/index.html.

8. ***, REST API Tutorial. *What is REST?* http://www.restapitutorial.com/lessons/whatisrest.html.

9. ***, UAIC, FII, Sabin-Corneliu Buraga. *Web Application Development*. http://profs.info.uaic.ro/~busaco/teach/courses/wade/.

10. ***, Scotch.io. *Build a RESTful API Using Node and Express 4.* http://scotch.io/tutorials/javascript/build-a-restful-api-using-node-and-express-4.

11. ***, *mongoose elegant mongodb object modeling for node.js*. http://mongoosejs.com/.

12. ***, Express. *Fast, unopinionated, minimalist web framework for Node.js*. http://expressjs.com.