RESTful Web API Design with Node.js

In this era of cloud computing, every data provisioning solution is built in a scalable and fail-safe way. Thus, when building RESTful services, the right choice of the underlying platform is vital. Node.js, with its asynchronous, event-driven architecture, is just the right choice to build RESTful APIs.

This book will be your step-by-step guide in the RESTful world of Node.js. It starts with the implementation of a simple HTTP handler application and follows its evolution to a completed RESTful service solution, which introduces you to the best practices in developing RESTful services.

Who this book is written for

If you are a web developer wanting to enrich your development skills to create scalable, server-side, RESTful applications based on the Node.js platform, this book is for you. You also need to be aware of HTTP communication concepts and should have a working knowledge of JavaScript. Knowledge of REST would be an added advantage but is definitely not a necessity.

What you will learn from this book

- Learn about NoSQL data stores and integrate MongoDB in your Node.js application with Mongoose
- Install, develop, and test your own Node.js user modules
- Eliminate third-party dependencies in your tests with mocking
- Comprehend the differences between an HTTP and RESTful application
- Optimize RESTful service URI routing with best practices
- Secure your services with NoSQL database integration within Node.js applications


Valentin Bojinov

Community Experience Distilled

RESTful Web API Design with Node.js

Design and implement comprehensive RESTful solutions in Node.js

$ 29.99 US
£ 19.99 UK

Prices do not include local sales tax or VAT
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'REST – What You Didn't Know'
- A synopsis of the book’s content
- More information on RESTful Web API Design with Node.js

About the Author

Valentin Bojinov studied computer programming at Technological School of Electronic Systems, a college within the Technical University of Sofia, Bulgaria. There, he got acquainted with programming and realized that his career would be in research and development. After college, he graduated from the Technical University of Sofia with a BSc in telecommunication and information engineering. It was then that his interest in data transmission grew and he ventured into B2B (business-to-business) communication. He is currently working on his MSc degree in software development. He is an expert in Java, SOAP, RESTful web services, and B2B integration.

A few years after Valentin started his career as a .NET developer, he realized that B2B and SOA were his passion. He then moved to SAP, where he contributed to the development of the web services stack of the SAP JEE platform. He currently works as a senior Java developer for the Bulgarian branch of Seeburger AG, a leader in the B2B and MFT solutions market. There, he develops and maintains several B2B communication adapters, including web services and SAP adapters.
RESTful Web API Design with Node.js

RESTful services have become the de facto standard data feed providers for social services, news feeds, and mobile devices. They deliver a large amount of data to millions of users. Thus, they need to address high-availability requirements, such as reliability and scalability. This book will show you how to utilize the Node.js platform to implement a robust and performant data service. By the end of this book, you will have learned how to implement a real-life RESTful service, taking advantage of the modern NoSQL database to serve both JSON and binary content.

Important topics such as correct URI structuring and security features are also covered, with detailed examples, showing you everything you need to know to start implementing the robust RESTful APIs that serve content for your applications.

What This Book Covers

Chapter 1, REST – What You Didn't Know, introduces the history of REST and how it couples with the HTTP protocol.

Chapter 2, Getting Started with Node.js, teaches you how to install Node.js and work with its package manager to install modules. It is in this chapter that you develop your first HTTP server application.

Chapter 3, Building a Typical Web API, takes you through structuring your application using human-readable URL and URI parameters. You get to develop a read-only RESTful service application using the filesystem for storage.

Chapter 4, Using NoSQL Databases, showcases how to use the LevelDB and MongoDB NoSQL databases, and explains the difference between key-value and document data stores.

Chapter 5, Implementing a Full-featured RESTful Service, takes you through implementing a production-ready RESTful service that uses NoSQL to store its data. You get to learn how to handle binary data and how to version an API while it evolves.

Chapter 6, Keeping the Bad Guys Out, covers restricting access to your data by choosing an appropriate authentication approach. Now you can protect data leakage with transport layer security.
Nowadays, topics such as cloud computing and mobile device service feeds, and other data sources being powered by cutting-edge, scalable, stateless, and modern technologies such as RESTful web services, leave the impression that REST has been invented recently. Well, to be honest, it is definitely not! In fact, REST was defined at the end of the 20th century.

This chapter will walk you through REST's history and will teach you how REST couples with the HTTP protocol. You will look at the five key principles that need to be considered while turning an HTTP application into a RESTful-service-enabled application. You will also look at the differences between RESTful and SOAP-based services. Finally, you will learn how to utilize already existing infrastructure for your benefit.

In this chapter, we will cover the following topics:

- A brief history of REST
- REST with HTTP
- RESTful versus SOAP-based services
- Taking advantage of existing infrastructure
A brief history of REST

Let's look at a time when the madness around REST made everybody talk restlessly about it! This happened back in 1999, when a request for comments was submitted to the Internet Engineering Task Force (IETF: http://www.ietf.org/) via RFC 2616: "Hypertext Transfer Protocol - HTTP/1.1." One of its authors, Roy Fielding, later defined a set of principles built around the HTTP and URI standards. This gave birth to REST as we know it today.


Let's look at the key principles around the HTTP and URI standards, sticking to which will make your HTTP application a RESTful-service-enabled application:

- Everything is a resource
- Each resource is identifiable by a unique identifier (URI)
- Use the standard HTTP methods
- Resources can have multiple representations
- Communicate statelessly

Principle 1 – everything is a resource

To understand this principle, one must conceive the idea of representing data by a specific format and not by a physical file. Each piece of data available on the Internet has a format that could be described by a content type. For example, JPEG Images; MPEG videos; html, xml, and text documents; and binary data are all resources with the following content types: image/jpeg, video/mpeg, text/html, text/xml, and application/octet-stream.
Principle 2 – each resource is identifiable by a unique identifier

Since the Internet contains so many different resources, they all should be accessible via URIs and should be identified uniquely. Furthermore, the URIs can be in a human-readable format (frankly I do believe they should be), despite the fact that their consumers are more likely to be software programmers rather than ordinary humans.

The URI keeps the data self-descriptive and eases further development on it. In addition, using human-readable URIs helps you to reduce the risk of logical errors in your programs to a minimum.

Here are a few sample examples of such URIs:


These human-readable URIs expose different types of resources in a straightforward manner. In the example, it is quite clear that the resource types are as follows:

- Images
- Videos
- XML documents
- Some kinds of binary archive documents

Principle 3 – use the standard HTTP methods

The native HTTP protocol (RFC 2616) defines eight actions, also known as verbs:

- GET
- POST
- PUT
- DELETE
- HEAD
- OPTIONS
- TRACE
- CONNECT
The first four of them feel just natural in the context of resources, especially when defining actions for resource data manipulation. Let's make a parallel with relative SQL databases where the native language for data manipulation is CRUD (short for Create, Read, Update, and Delete) originating from the different types of SQL statements: INSERT, SELECT, UPDATE and DELETE respectively. In the same manner, if you apply the REST principles correctly, the HTTP verbs should be used as shown here:

<table>
<thead>
<tr>
<th>HTTP verb</th>
<th>Action</th>
<th>Response status code</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Request an existing resource</td>
<td>&quot;200 OK&quot; if the resource exists, &quot;404 Not Found&quot; if it does not exist, and &quot;500 Internal Server Error&quot; for other errors</td>
</tr>
<tr>
<td>PUT</td>
<td>Create or update a resource</td>
<td>&quot;201 CREATED&quot; if a new resource is created, &quot;200 OK&quot; if updated, and &quot;500 Internal Server Error&quot; for other errors</td>
</tr>
<tr>
<td>POST</td>
<td>Update an existing resource</td>
<td>&quot;200 OK&quot; if the resource has been updated successfully, &quot;404 Not Found&quot; if the resource to be updated does not exist, and &quot;500 Internal Server Error&quot; for other errors</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete a resource</td>
<td>&quot;200 OK&quot; if the resource has been deleted successfully, &quot;404 Not Found&quot; if the resource to be deleted does not exist, and &quot;500 Internal Server Error&quot; for other errors</td>
</tr>
</tbody>
</table>

There is an exception in the usage of the verbs, however. I just mentioned that POST is used to create a resource. For instance, when a resource has to be created under a specific URI, then PUT is the appropriate request:

```
PUT /data/documents/balance/22082014 HTTP/1.1
Content-Type: text/xml
Host: www.mydatastore.com

<?xml version="1.0" encoding="utf-8"?>
<balance date="22082014">
  <Item>Sample item</Item>
  <price currency="EUR">100</price>
</balance>
```

HTTP/1.1 201 Created
Content-Type: text/xml
Location: /data/documents/balance/22082014
However, in your application you may want to leave it up to the server REST
application to decide where to place the newly created resource, and thus create it
under an appropriate but still unknown or non-existing location.

For instance, in our example, we might want the server to create the date part of the
URI based on the current date. In such cases, it is perfectly fine to use the POST verb
to the main resource URI and let the server respond with the location of the newly
created resource:

```markdown
POST /data/documents/balance HTTP/1.1
Content-Type: text/xml
Host: www.mydatastore.com

<?xml version="1.0" encoding="utf-8"?>
<balance date="22082014">
  <Item>Sample item</Item>
  <price currency="EUR">100</price>
</balance>

HTTP/1.1 201 Created
Content-Type: text/xml
Location: /data/documents/balance
```

**Principle 4 – resources can have multiple representations**

A key feature of a resource is that they may be represented in a different form than
the one it is stored. Thus, it can be requested or posted in different representations.
As long as the specified format is supported, the REST-enabled endpoint should use
it. In the preceding example, we posted an xml representation of a balance, but if the
server supported the JSON format, the following request would have been valid as
well:

```markdown
POST /data/documents/balance HTTP/1.1
Content-Type: application/json
Host: www.mydatastore.com

{
  "balance": {
    "date": "22082014",
    "Item": "Sample item",
    "price": {
```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /data/documents/balance

Principle 5 – communicate statelessly

Resource manipulation operations through HTTP requests should always be considered atomic. All modifications of a resource should be carried out within an HTTP request in isolation. After the request execution, the resource is left in a final state, which implicitly means that partial resource updates are not supported. You should always send the complete state of the resource.

Back to the balance example, updating the price field of a given balance would mean posting a complete JSON document that contains all of the balance data, including the updated price field. Posting only the updated price is not stateless, as it implies that the application is aware that the resource has a price field, that is, it knows its state.

Another requirement for your RESTful application is to be stateless; the fact that once deployed in a production environment, it is likely that incoming requests are served by a load balancer, ensuring scalability and high availability. Once exposed via a load balancer, the idea of keeping your application state at server side gets compromised. This doesn’t mean that you are not allowed to keep the state of your application. It just means that you should keep it in a RESTful way. For example, keep a part of the state within the URI.

The statelessness of your RESTful API isolates the caller against changes at the server side. Thus, the caller is not expected to communicate with the same server in consecutive requests. This allows easy application of changes within the server infrastructure, such as adding or removing nodes.

Remember that it is your responsibility to keep your RESTful APIs stateless, as the consumers of the API would expect it to be.
Now that you know that REST is around 15 years old, a sensible question would be, "why has it become so popular just quite recently?" My answer to the question is that we as humans usually reject simple, straightforward approaches, and most of the time, we prefer spending more time on turning complex solutions into even more complex and sophisticated solutions.

Take classical SOAP web services for example. Their various WS-* specifications are so many and sometimes loosely defined in order to make different solutions from different vendors interoperable. The WS-* specifications need to be unified by another specification, WS-BasicProfile.

This mechanism defines extra interoperability rules in order to ensure that all WS-* specifications in SOAP-based web services transported over HTTP provide different means of transporting binary data. This is again described in other sets of specifications such as SOAP with Attachment References (SwaRef) and Message Transmission Optimisation Mechanism (MTOM), mainly because the initial idea of the web service was to execute business logic and return its response remotely, not to transport large amounts of data.

Well, I personally think that when it comes to data transfer, things should not be that complex. This is where REST comes into place by introducing the concept of resources and standard means to manipulate them.

**The REST goals**

Now that we've covered the main REST principles, let's dive deeper into what can be achieved when they are followed:

- Separation of the representation and the resource
- Visibility
- Reliability
- Scalability
- Performance
Separation of the representation and the resource

A resource is just a set of information, and as defined by principle 4, it can have multiple representations. However, the state of the resource is atomic. It is up to the caller to specify the content-type header of the http request, and then it is up to the server application to handle the representation accordingly and return the appropriate HTTP status code:

- HTTP 200 OK in the case of success
- HTTP 400 Bad request if a unsupported content type is requested, or for any other invalid request
- HTTP 500 Internal Server error when something unexpected happens during the request processing

For instance, let’s assume that at the server side, we have balance resources stored in an XML file. We can have an API that allows a consumer to request the resource in various formats, such as application/json, application/zip, application/octet-stream, and so on.

It would be up to the API itself to load the requested resource, transform it into the requested type (for example, json or xml), and either use zip to compress it or directly flush it to the HTTP response output. It is the Accept HTTP header that specifies the expected representation of the response data. So, if we want to request our balance data inserted in the previous section in XML format, the following request should be executed:

GET /data/balance/22082014 HTTP/1.1
Host: my-computer-hostname
Accept: text/xml

HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 140

<?xml version="1.0" encoding="utf-8"?>
<balance date="22082014">
<Item>Sample item</Item>
<price currency="EUR">100</price>
</balance>
To request the same balance in JSON format, the `Accept` header needs to be set to `application/json`:

```
GET /data/balance/22082014 HTTP/1.1
Host: my-computer-hostname
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 120

{
  "balance": {
    "date": "22082014",
    "Item": "Sample item",
    "price": {
      "-currency": "EUR",
      "#text": "100"
    }
  }
}
```

Visibility

REST is designed to be visible and simple. Visibility of the service means that every aspect of it should be self-descriptive and should follow the natural HTTP language according to principles 3, 4, and 5.

Visibility in the context of the outer world would mean that monitoring applications would be interested only in the HTTP communication between the REST service and the caller. Since the requests and responses are stateless and atomic, nothing more is needed to flow the behavior of the application and to understand whether anything has gone wrong.

Remember that caching reduces the visibility of your RESTful applications and should be avoided.
Reliability

Before talking about reliability, we need to define which HTTP methods are safe and which are idempotent in the REST context. So let’s first define what safe and idempotent methods are:

- An HTTP method is considered to be safe provided that when requested, it does not modify or cause any side effects on the state of the resource
- An HTTP method is considered to be idempotent if its response is always the same, no matter how many times it is requested

The following table lists shows you which HTTP method is safe and which is idempotent:

<table>
<thead>
<tr>
<th>HTTP Method</th>
<th>Safe</th>
<th>Idempotent</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>POST</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PUT</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>DELETE</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Scalability and performance

So far, I have often stressed on the importance of having stateless implementation and stateless behavior for a RESTful web application. The World Wide Web is an enormous universe, containing a huge amount of data and a few times more users eager to get that data. Its evolution has brought about the requirement that applications should scale easily as their load increases. Scaling applications that have a state is hardly possible, especially when zero or close-to-zero downtime is needed.

That’s why being stateless is crucial for any application that needs to scale. In the best-case scenario, scaling your application would require you to put another piece of hardware for a load balancer. There would be no need for the different nodes to sync between each other, as they should not care about the state at all. Scalability is all about serving all your clients in an acceptable amount of time. Its main idea is keep your application running and to prevent Denial of Service (DoS) caused by a huge amount of incoming requests.

Scalability should not be confused with performance of an application. Performance is measured by the time needed for a single request to be processed, not by the total number of requests that the application can handle. The asynchronous non-blocking architecture and event-driven design of Node.js make it a logical choice for implementing a well-scalable application that performs well.
Working with WADL

If you are familiar with SOAP web services, you may have heard of the **Web Service Definition Language (WSDL)**. It is an XML description of the interface of the service. It is mandatory for a SOAP web service to be described by such a WSDL definition.

Similar to SOAP web services, RESTful services also offer a description language named WADL. WADL stands for Web Application Definition Language. Unlike WSDL for SOAP web services, a WADL description of a RESTful service is optional, that is, consuming the service has nothing to do with its description.

Here is a sample part of a WADL file that describes the GET operation of our balance service:

```xml
<application xmlns="http://wadl.dev.java.net/2009/02"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:service="http://localhost:8080/data/balance">
  <grammar>
    <include href="balance.xsd"/>
    <include href="error.xsd"/>
  </grammar>
  <resources base="http://localhost:8080/data/balance/">
    <resource path="{date}">
      <method name="GET">
        <request>
          <param name="date" type="xsd:string" style="template"/>
        </request>
        <response status="200">
          <representation mediaType="application/xml" element="service:balance"/>
          <representation mediaType="application/json"/>
        </response>
        <response status="404">
          <representation mediaType="application/xml" element="service:balance"/>
        </response>
      </method>
    </resource>
  </resources>
</application>
```
This extract of a WADL file shows how application-exposing resources are described. Basically, each resource must be a part of an application. The resource provides the URI where it is located with the base attribute, and describes each of its supported HTTP methods in a method. Additionally, an optional doc element can be used at resource and application to provide additional documentation about the service and its operations.

Though WADL is optional, it significantly reduces the efforts of discovering RESTful services.

**Taking advantage of the existing infrastructure**

The best part of developing and distributing RESTful applications is that the infrastructure needed is already out there waiting restlessly for you. As RESTful applications use the existing web space heavily, you need to do nothing more than following the REST principles when developing. In addition, there are a plenty of libraries available out there for any platform, and I do mean any given platform. This eases development of RESTful applications, so you just need to choose the preferred platform for you and start developing.

**Summary**

In this chapter, you learned about the history of REST, and we made a slight comparison between RESTful services and classical SOAP Web services. We looked at the five key principles that would turn our web application into a REST-enabled application, and finally took a look at how RESTful services are described and how we can simplify the discovery of the services we develop.

Now that you know the REST basics, we are ready to dive into the Node.js way of implementing RESTful services. In the next chapter, you will learn about the essentials of Node.js and the accompanying tools that are necessary to use and understand in order to build a full-fledged web service.
Where to buy this book

You can buy RESTful Web API Design with Node.js from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.