CHAPTER 4

Architecting a REST API

It is extremely important to understand a REST-based architecture, meaning how the system will look if you're basing all of your services in the REST style. But it is equally important to know what the internal architecture of those REST services will look like before you start working.

In Node.js there are several modules out there, with thousands of daily downloads that can help you create an API without having to worry too much about the internal aspects of it. And that might be a good idea if you're in a hurry to get the product out, but since you're here to learn, I'll go over all the components that make up a standard, general-purpose REST API.

The modules are mentioned, but I won't go into details on how they're used or anything; that will come in the next chapter— so keep reading!

For the purpose of this book, I'll take the traditional approach when it comes to architecting the API, and you'll use an MVC pattern (model–view–controller); although you might be familiar with other options, it is one of the most common ones and it normally fits well as a web use case.

The basic internal architecture of a RESTful API contains the following items:

- *A request handler*. This is the focal point that receives every request and processes it before doing anything else.
- *A middleware/pre-process chain*. These guys help shape the request and provide some help for authentication control.
- *A routes handler*. After the request handler is done, and the request itself has been checked and enriched with everything you need, this component figures out who needs to take care of the request.
- *The controller*. This guy is responsible for all requests done related to one specific resource.
- *The Model*. Also known as the *resource* in our case. You'll focus most of the logic related to the resource in here.
- *The representation layer*. This layer takes care of creating the representation that is visible to the client app.
- *The response handler*. Last but certainly not least, the response handler takes care of sending the representation of the response back to the client.

Note As I've stated several times before, this book focuses on HTTP-based REST, which means that any request mentioned in this chapter is an HTTP request, unless otherwise stated.

F. Doglio, *Pro REST API Development with Node.js*, DOI 10.1007/978-1-4842-0917-2_4

The Request Handler, the Pre-Process Chain, and the Routes Handler

The *request handler*, the *pre-process chain*, and the *routes handler* are the first three components in any request to your system, so they're key to having a responsive and fast API. Luckily, you're using Node.js, and as you saw in Chapter 3, Node.js is great at handling many concurrent requests because of its event loop and async I/O.

That being said, let's list the attributes our request handler needs to have for our RESTful system to work as expected:

- It has to gather all the HTTP headers and the body of the request, parse them, and provide a request object with that information.
- It needs to be able to communicate with both the pre-processing chain module and the routes handler in order to figure out which controller needs to be executed.
- It needs to create a response object capable of finishing and (optionally) writing a response back to the client.

Figure 4-1 shows the steps that are part of the initial contact between client and server:

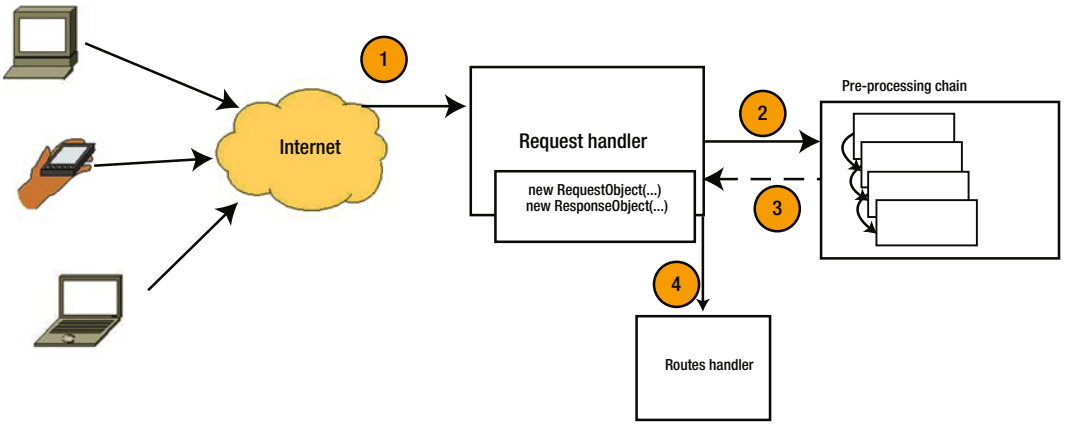

Figure 4-1. *Example of how the request handler and its interactions with other components look*

1. The client application issues a request for a particular resource.
2. The request handler gathers all information. It creates a request object and passes it along to the pre-processing chain.
3. Once finished, the pre-processing chain returns the request object—with whatever changes made to it—to the request handler.
4. Finally, the RH sends the request and response objects to the routes handler so that the process can continue.

There is one problem in Figure 4-1 that jumps right out at you (or it should): if the pre-processing chain takes too long, the request handler must wait for it to finish before handing over the request to the routes handler, and any other incoming request is forced to wait as well.

This can be especially harmful to the performance of the API if the pre-processing chain is doing some heavy-duty tasks, like loading user-related data or querying external services.

Thanks to the fact that you're using Node.js as the basis for everything here, you can easily change the pre-processing chain to be an asynchronous operation. By doing that, the request handler is able to receive new requests while still waiting for the processing chain from the previous request. Figure 4-2 shows how the diagram would change.

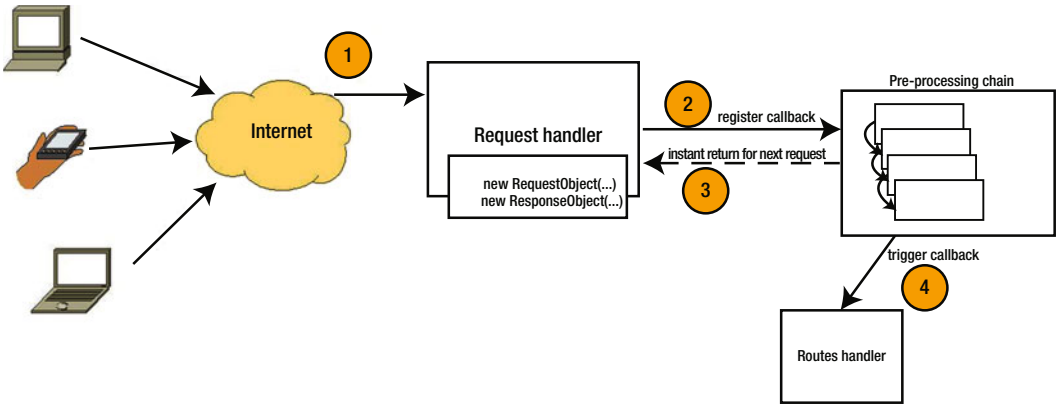

Figure 4-2. *Changes to the architecture show how to improve it by using a callback*

As you can see in Figure 4-2, the change is minimal, at least at the architectural level. The request handler sets up a callback to the routes handler; and that callback is executed once the pre-processing chain is finished. And right after setting up the callback, the request handler is free again for the next request. This clearly provides more freedom to this component, allowing the entire system to process more requests per second.

■ **Note** This change will not actually speed up the pre-processing chain time of execution, neither will it speed up the time it takes a single request to be finished, but it will allow the API to handle more requests per second, which in practice means avoiding an obvious bottleneck.

As for the pre-processing chain, you'll use it for generic operations, things that are required in most of the routes you'll handle. That way you can extract that code from the handlers and centralize it into small units of code (functions) that are called in sequence for every request.

Most of the modules you'll see in the next chapter have one version of the pre-processing chain. For instance, Express.js calls the functions that can be executed in the chain "middleware." Vatican.js calls them "pre-processors" to differentiate them from the post-processors that the module also provides.

■ **Tip** The main rule to remember when adding a new function into this chain is that as a good practice, the function should take care of one task, and one task only. (This is generally a good practice to follow on every aspect of software development, some call it the Unix Philosophy, others call it KISS; call it whatever you want, it's a good idea to keep in mind.) That way, it becomes mind-blowingly easy to enable and disable them when testing, even to alter their order. On the other hand, if you start adding functions that take care of more than one thing, like authenticating the user and loading his/her preferences, you'll have to edit the function's code to disable one of those services.

Since you'll want the entire pre-processing to be done asynchronously to release the request handler from waiting for the chain to be done, the chain will use asynchronous serial flow. This way you can be sure of the order of execution; but at the same time, you're free to have these functions perform actions that take longer than normal, like asynchronous calls to external services, I/O operations, and the like.

Let's take a final look at the last diagram. So far it looks great: you're able to handle requests asynchronously and you can do some interesting things to the request by pre-processing it before giving it to the routes handler. But there is one catch: the pre-processing chain is the same for all routes.

That might not be a problem if the API is small enough, but just to be on the safe side, and to provide a fully scalable architecture, let's take a look at another change that can be done over the current version to provide the freedom you require (see Figure 4-3).

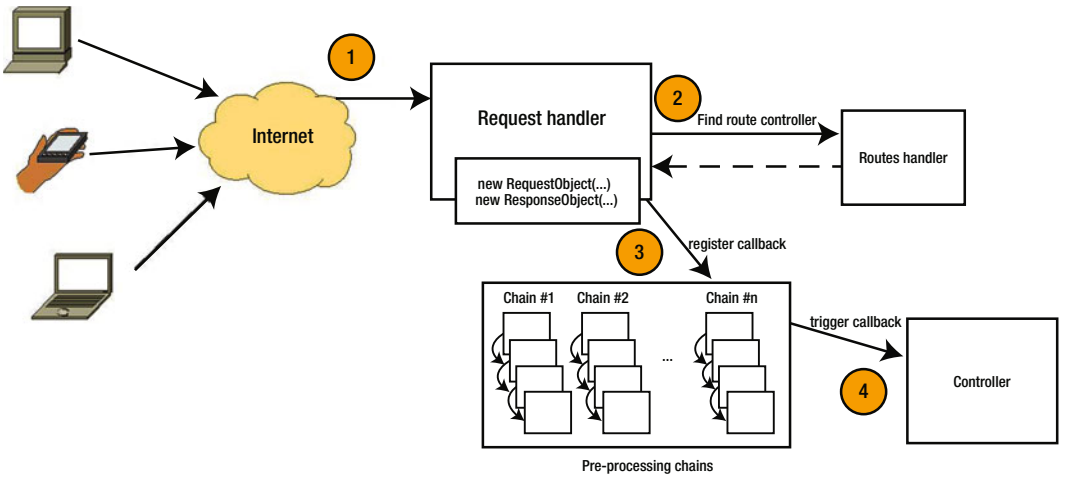

Figure 4-3. *Change on the architecture to provide room for multiple pre-processing chains*

This chain (as shown in Figure 4-3) is bigger than the previous one, but it does solve the scaling problem. The process has now changed into the following steps:

1. The client application issues a request for a particular resource.
2. The request handler gathers all information. It creates a request object and passes it along to the request handler to return the right controller. This action is simple enough to do synchronously. Ideally, it should be done in constant time (O(1)).

3. Once it has the controller, it registers an asynchronous operation with the correct pre-processing chain. This time around, the developer is able to set up as many chains as needed and associates them to one specific route. The request handler also sets up the controller's method to be executed as the callback to the chain's process.
4. Finally, the callback is triggered, and the request object, with the response object passed into the controller's method to continue the execution.

■ **Note** Step 2 mentions that the controller lookup based on the request should be done in constant time. This is not a hard requirement, but should be the desirable result; otherwise, when handling many concurrent requests, this step might become a bottleneck that can affect subsequent requests.

MVC: a.k.a. Model–View–Controller

The model–view– controller (MVC) architectural pattern[1] is probably *the most well-known pattern out there*. Forget about the Gang of Four's design patterns,[2] forget about everything you learned about software design and architectural patterns; if you're comfortable with MVC, you have nothing to worry about.

Actually, that's not true; well, most of it isn't anyway. MVC *is* currently among the most well-known and used design patterns on web projects (that much is true). That being said, you should not forget about the others; in fact, I highly recommend you actually get familiar with the most common ones (aside from MVC of course), like Singleton, Factory, Builder, Adapter, Composite, Decorator, and so forth. Just look them up, and read and study some examples; it's always handy to have them as part of your tool box.

Going back to MVC, even though it has become really popular in the last few years, especially since 2007 (coincidently the year when version 2 of Ruby on Rails , a popular web framework that had MVC as part of its core architecture, was released), this bad boy is not new. In fact, it was originally described by Krasner and Pope in 1988 at SmallTalk-80[3] as a design pattern for creating user interfaces.

The reason why it is such a popular pattern on web projects is because it fits perfectly into the multilayer architecture that the web provides. Think about it: due to the client-server architecture, you already have two layers there, and if you organized code to split some responsibilities between orchestration and business logic, you gain one more layer on the server side, which could translate into the scenario shown in Table 4-1.

Table 4-1. *List of Layers*

Layer	Description
Business logic	You can encapsulate the business logic of the system into different components, which you can call *models*. They represent the different resources that the system handles.
Orchestration	The models know how to do their job, but not when or what kind of data to use. The controllers take care of this.
Representation layer	Handles creating the visual representation of the information. In a normal web application, this is the HTML page. In a RESTful API, this layer takes care of the different representations each resource has.

[1]See http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller.
[2]See http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/.
[3]See http://dl.acm.org/citation.cfm?id=50757.50759.

■ **Note** Prior to Table 4-1, I mentioned that the client-server architecture provided the first two layers for MVC, meaning that the client would act as the presentation layer. This is not entirely true, as you see later on, but it does serve as a conceptual layer, meaning that you'll need a way for the application to present the information to the user (client).

Let's look at the diagram in Figure 4-4, which represents Table 4-1.

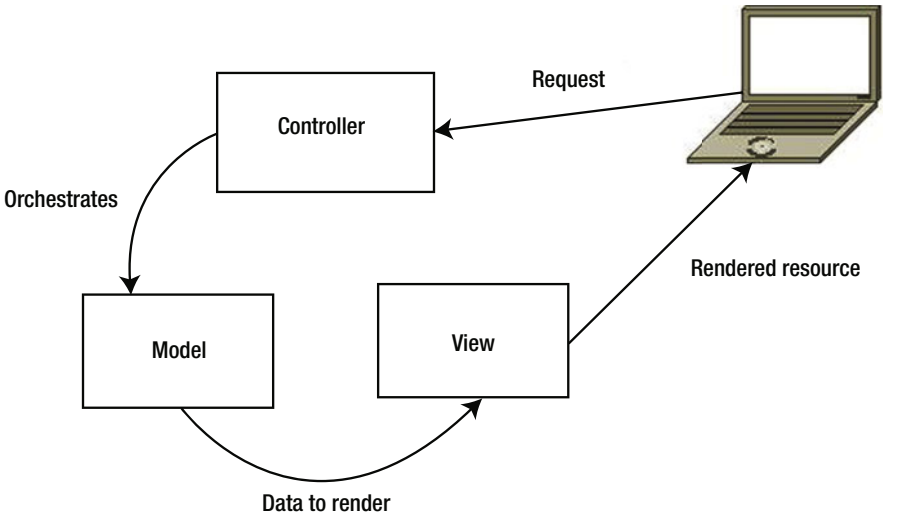

Figure 4-4. *The interaction between the three layers*

Figure 4-4 shows the decoupling of the three components: the controller, the model (which in this case you can also call the resource), and the view. This decoupling allows for a clear definition of each component's responsibilities, which in turn helps keep the code clean and easy to understand.

Although this is great, the pattern has changed a bit ever since it was adopted by some web development frameworks, like Ruby on Rails; it now looks more like what's shown in Figure 4-5.

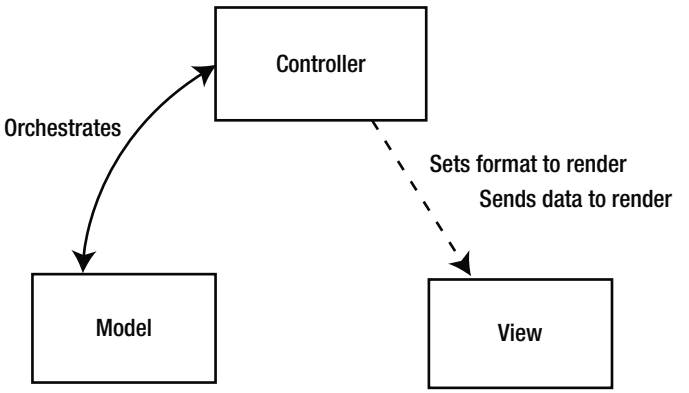

Figure 4-5. *MVC applied to the web*

The current iteration of the pattern removed the relationship between the model and the view, and instead gave the controller that responsibility. The controller now also orchestrates the view.

This final version is the one you'll add into our current growing architecture. Let's take a look at how it will look (see Figure 4-6).

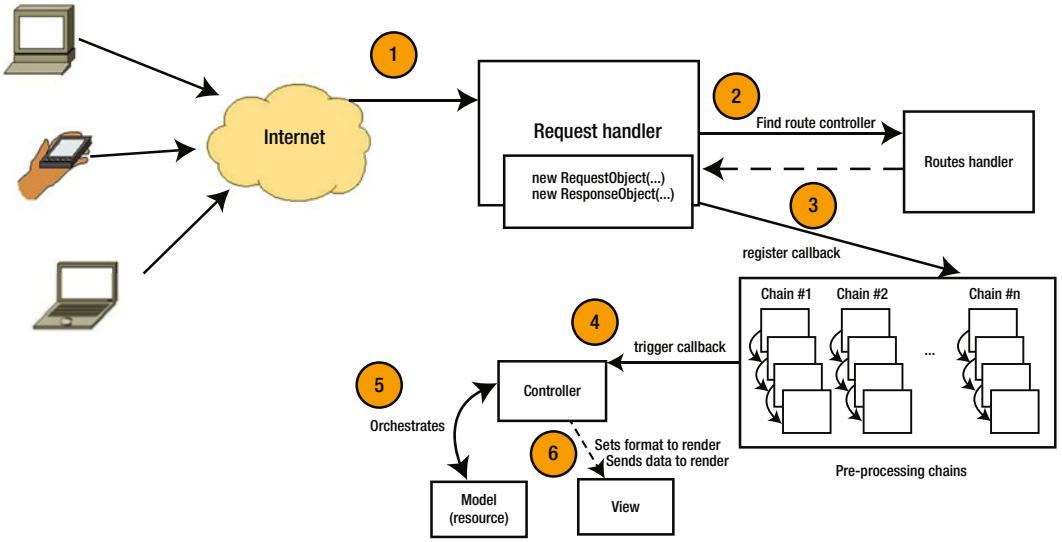

Figure 4-6. *The architecture with the added MVC pattern*

Steps 5 and 6 have been added to our architecture. When the right method on the controller is triggered (in step 4), it handles interacting with the model, gathers the required data, and then sends it over to the view to render it back to the client application.

This architecture works great, but there is still one improvement that can be done. With our RESTful API, the representations are strictly related to the resources data structure, and you can generalize the view into a view layer, which will take care of transforming the resources into whatever format you require. This change simplifies the development since you centralize the entire view-related code into one single component (the view layer).

The diagram in Figure 4-7 might not have changed a lot, but the change in the view box into a view layer represents the generalization of that code, which initially implied that there would be one specific view code for every resource.

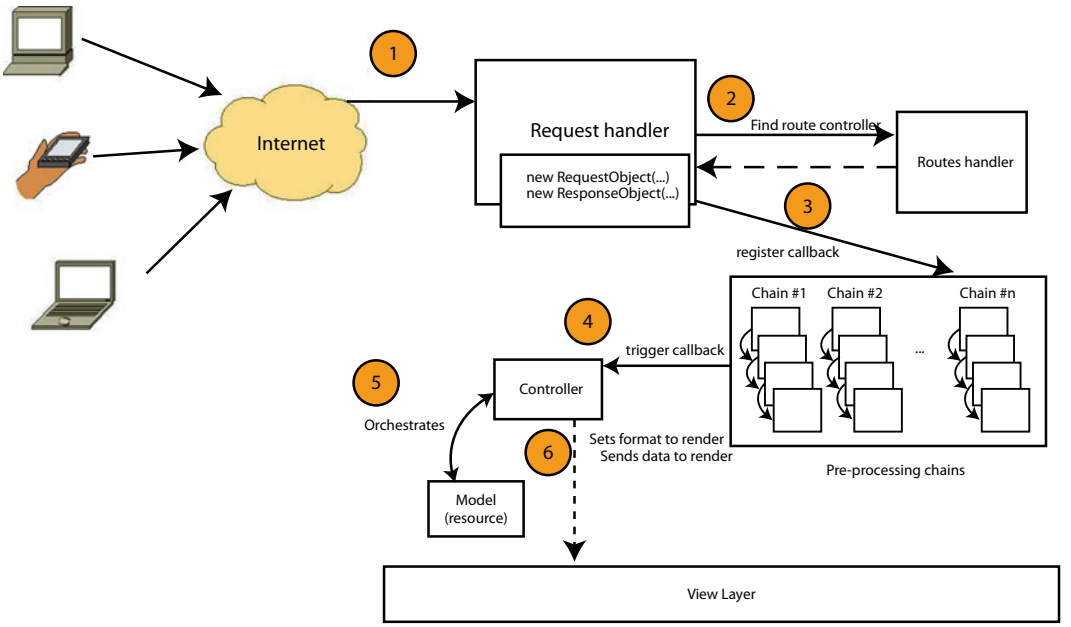

Figure 4-7. *View layer added to the architecture*

Alternatives to MVC

MVC is a great architecture. Nearly every developer is using it or has used it for a web project. Of course, not everyone loves it, because it has suffered the same fate other popular things in the development community have suffered (Ruby on Rails anyone?). If it becomes popular on the Internet, everyone is using it for everything—until they realize that not every project looks like an MVC nail, so you have to start looking at other shapes of hammers (other alternatives architectures).

But luckily, there are alternatives; there are similar architectural patterns that may better suit your needs, depending on the particular aspects of your project. Some of them are direct derivatives of MVC, and others try to approach the same problem from a slightly different angle (I say "slightly" because, as you're about to see, there are things in common).

Hierarchical MVC

Hierarchical MVC[4] is a more complex version of MVC in the sense that you can nest one MVC component inside another one. This gives developers the ability to have things like an MVC group for a page, another MVC group for the navigation inside the page, and a final MVC component for the contents of the page.

This approach is especially helpful when developing reusable widgets that can be plugged into components, since each MVC group is self-contained. It is useful in cases when the data to be displayed comes from different related sources. In these cases, having a HMVC structure helps keep the separation of concerns intact, and avoids coupling between components that shouldn't be.

[4]See http://en.wikipedia.org/wiki/Hierarchical_model%E2%80%93view%E2%80%93controller.

Let's look at a very basic example. Think of a user reading a blog post and the related comments underneath it. There are two ways to go about it: with MVC or with HMVC.

With MVC, the request is done to the BlogPosts controller, since that is the main resource being requested; afterward, that controller loads the proper blog post model and using that model's ID, it loads the related comments models. Right there, there is an unwanted coupling between the BlogPosts controller and the comments model. You can see this in the diagram in Figure 4-8.

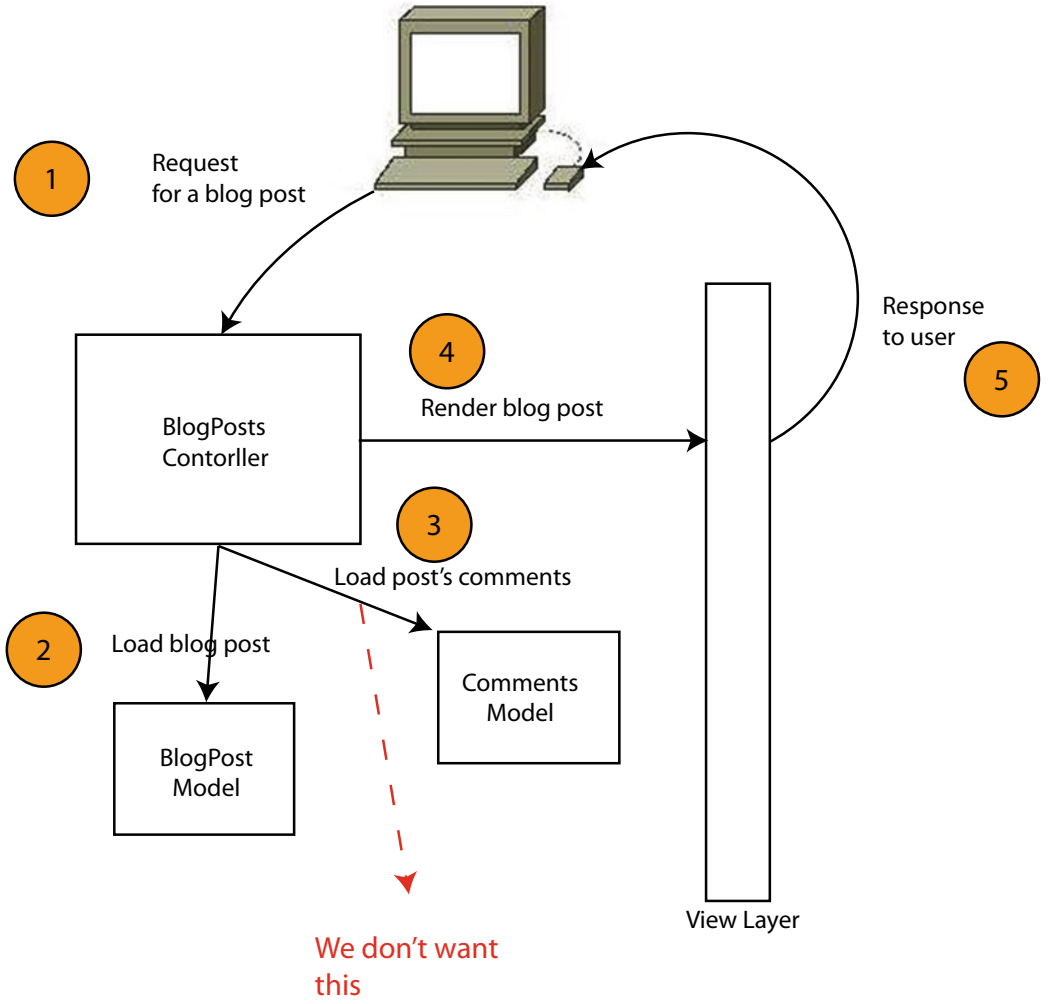

Figure 4-8. *The problem that HMVC tries to solve*

Figure 4-8 shows the coupling that you need to get rid of; it is clearly something that can be improved from an architectural point of view. So let's look at what this would look like using HMVC (see Figure 4-9).

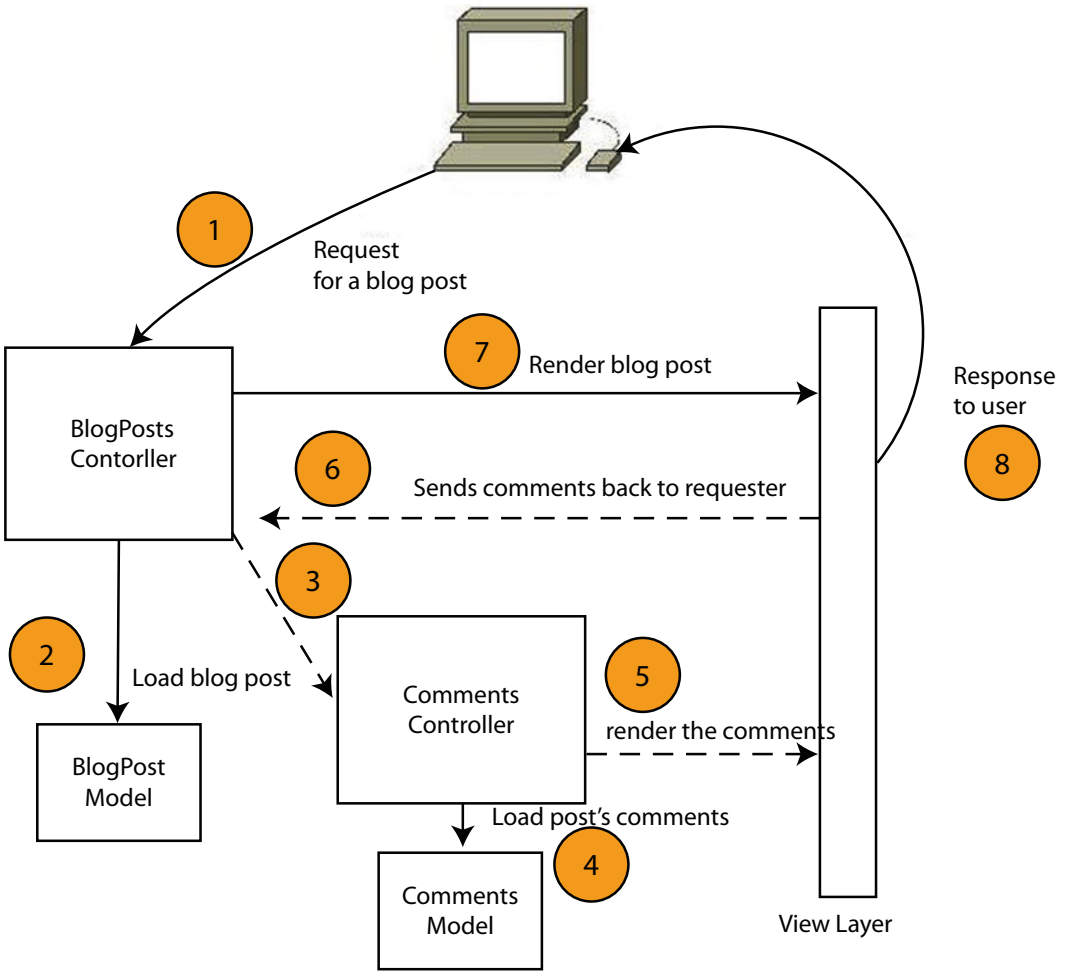

Figure 4-9. *The same diagram with the HMVC pattern applied*

The architecture certainly looks more complex and there are more steps, but it is also cleaner and easier to extend. Now in step 3, you're sending a request to an entirely new MVC component, one in charge of dealing with comments. That component will in turn interact with the corresponding model and with the generic view layer to return the representation of the comments. The representation is received by the BlogPost controller, which attaches it to the data obtained from the BlogPost model and sends everything back into the view layer.

If you want to create a new section in the blog showing specific blog posts and their comments, you could easily reuse the comments component.

All in all, this pattern could be considered a specialization of common MVC, and it could come in handy when designing complex systems.

Model–View–ViewModel

The Model–View–ViewModel pattern[5] was created by Microsoft in 2005 as a way to facilitate UI development using WPF and Silverlight; it allows UI developers to write code using a markup language (called XAML) focusing on the User Experience (UX), and accessing the dynamic functionalities using bindings to the code. This approach allows developers and UX developers to work independently without affecting each other's work.

Just like with MVC, the Model in this architecture concentrates the business logic, while the ViewModel acts as a mediator between the Model and the View, exposing the data from the first one. It also contains most of the view logic, allowing the ViewLayer to only focus on displaying information, leaving all dynamic behavior to the ViewModel.

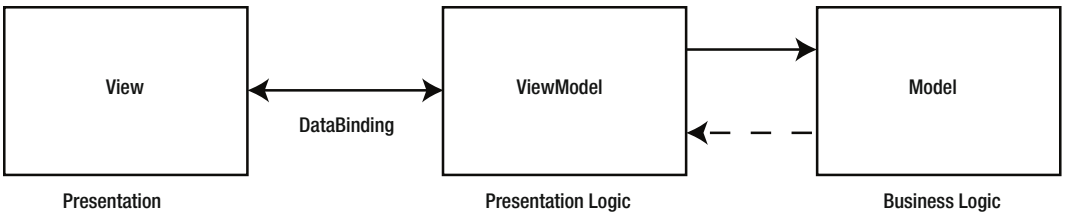

Figure 4-10. *An MVVC architecture*

These days, the pattern has been adopted by others outside Microsoft, like the ZK framework in Java and KnockoutJS, AngularJS, Vue.js, and other frameworks in JavaScript (since MVVM is a pattern specializing in UI development, it makes sense that UI frameworks written in JavaScript are big adopters of this pattern).

Model–View–Adapter

The model–view–adapter[6] (MVA) pattern is very similar to MVC, but with a couple of differences. Mainly, in MVC the main business logic is concentrated inside each model, which also contains the main data structure, with the controller in charge of orchestrating the model and the view.

In MVA, the model is just the data that you're working with, and the business logic is concentrated in the adapter, which is in charge of interacting both with the view and the model. So basically, slimmer models and fatter controllers. But joking aside, this allows for a total decoupling of the view and the model, giving all responsibilities to the adapter.

This approach works great when switching adapters to achieve different behaviors on the same view and model.

The architecture for this pattern is shown in Figure 4-11.

[5]See http://en.wikipedia.org/wiki/Model_View_ViewModel.
[6]See http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93adapter.

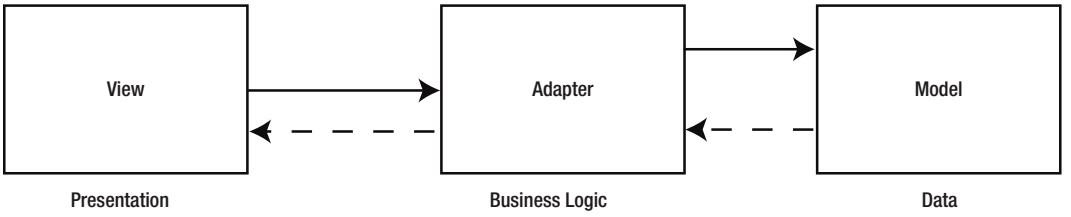

Figure 4-11. *The MVA pattern shown as a diagram*

Response Handler

The final component to our API architecture is the *response handler*; it is in charge of grabbing the resource representation from the view layer and sending it back to the client. The response format (which is not the same as the representation's format) must be the same as the request's format; in this case, it'll be an HTTP 1.1 message.

The HTTP response has two parts: the header, which contains several fields specifying properties about the message, and the body. The content of the message's body is the actual representation of the resource. The header is the section that interests us the most right now; it contains fields like content-type, content-length, and so on. Some of those fields are mandatory and some of them are required if you intend to follow the REST style fully (which you do).

- *Cacheable*: From the constraints imposed by REST defined in Chapter 1. Every request must be explicitly or implicitly set as cacheable when applicable. This translates into the use of the HTTP header `cache-control`.
- *Content-type*: The content type of the response's body is important for the client application to understand how to parse the data. If your resources only have one possible representation, the content type might be an optional header since you could notify the client app developer about the format through your documentation. But if you were to change it in the future, or add a new one, then it might cause some serious damage to your clients. So consider this header mandatory.
- *Status*: The status code is not mandatory but extremely important, as I've mentioned in previous chapters. It provides the client application a quick indicator of the result of the request.
- *Date*: This field should contain the date and time when the message was sent. It should be in HTTP-date format[7] (e.g., Fri, 24 Dec 2014 23:34:45 GMT).
- *Content-length*: This field should contain the number of bytes (length) of the body of the message transferred.

[7]See `http://tools.ietf.org/html/rfc7231#section-7.1.1.1`.

Let's look at an example of an HTTP response with the JSON representation of a resource:

```
HTTP/1.0 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: application/json
Cache-control: private, max-age=0, no-cache
Content-Length: 1354

{
  "name": "J.K.Rolling",
  "id": "ab12351bac",
  "books": [
     {
     "title": "Harry Potter and the Philosopher's Stone",
     "isbn": "9788478888566"
     },
     {
     "title": "Harry Potter and the Prisoner of Azkaban",
     "isbn": "9788422685227"
   }
 ]
}
```

There is one more improvement that could be made on the response handler if you want to get some extra juice. This is entirely extra, and most of the Node.js frameworks out there don't have it (with the exception of Vatican.js).

The idea is to have a post-processing chain of functions that receives the response content returned by the view layer, and transforms it, or enriches it if you will, with further data. It would act as the first version of the pre-processing chain: one common chain for the entire process.

With this idea, you can abstract further code from the controllers just by moving it into the post-processing stage. Code like schema validation (which I'll discuss later in the book) or response header setup can be centralized here, and with the added extra of a simple mechanism for switching it around or disabling steps in the chain.

Let's take a look at the final architecture of our API (see Figure 4-12).

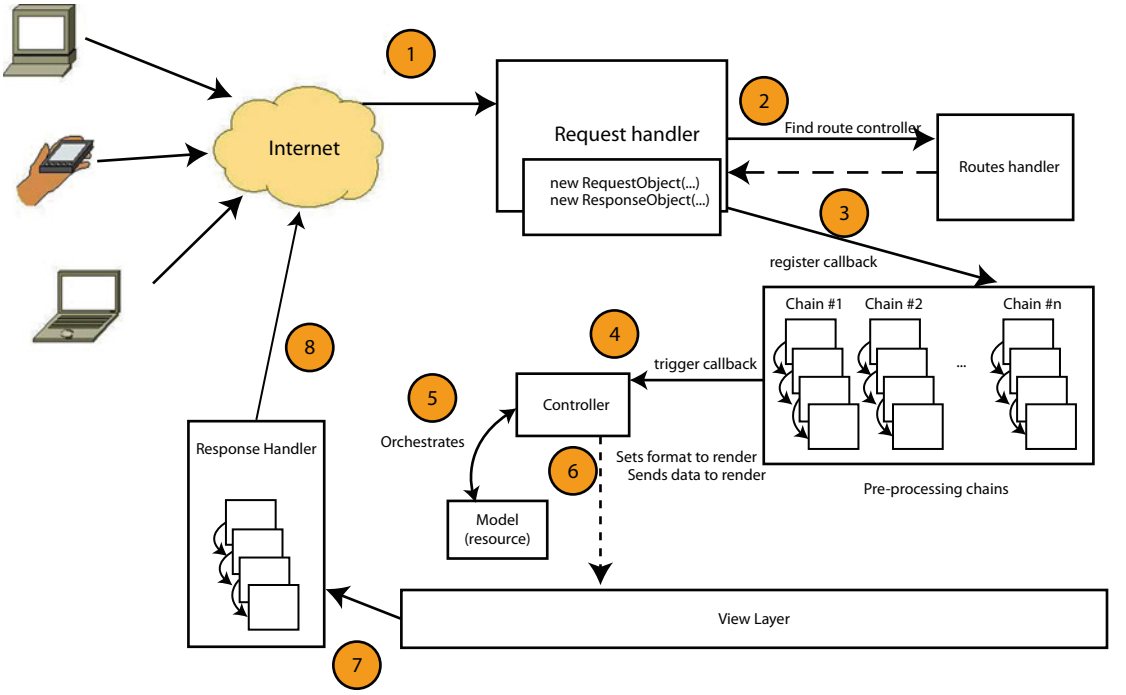

Figure 4-12. *The final architecture with the response handler and the added post-processing chain*

Summary

This chapter covered the basics for a complete and functional RESTful API architecture. It even covered some extras that aren't required but are certainly nice to have, such as pre- and post-processing. You also looked at the main architecture behind our design (MVC) and some alternatives to it, in case your requirements aren't a perfect match for the MVC model.

In the next chapter, I'll start talking about the modules you'll use to write the implementation of this architecture.