Author Picks

FREE

Understanding API Security

Chapters selected by
Justin Richer and Antonio Sanso

manning

Understanding API Security

With chapter selections by Justin Richer
and Antonio Sanso

Manning Author Picks

To pre-order or learn more about these books go to www.manning.com

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN 9781617294327
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 21 20 19 18 17 16

contents

introduction

We live in a programmable world. Every day new and inventive services come online, allowing us to connect our lives together like never before. Gone are the days when it was acceptable for a piece of software to live in its own little silo, disconnected from the outside world. Today, services are expected to be available for programming, mixing, and building into new applications.

The web-based Application Programming Interface, or API, is the means by which services make themselves available in this dynamic world. By exposing an API, a service can find new life and utility far beyond what its core functionality was designed for.

But it's not enough to just expose an API: these APIs need to be secured and protected in order to be truly useful. An API that's simply left open to everyone, with no security controls, cannot be used to protect personalized or sensitive information, which severely limits its desirability. There have been many approaches to this over the years, with many proprietary and bolted-on solutions having come and gone. The OAuth delegation and authorization protocol is one of the most popular standards for this today, replacing many of these hacks with a standard technology.

We've brought together several chapters from several Manning books that give you some context for how API security works in the real world by showing how APIs are put together and how the OAuth protocol can be used to protect them.

The OAuth Dance

OAuth 2.0 is a delegation and authorization security protocol. Unlike many other protocols, which are an end to themselves, the OAuth 2.0 protocol is always used in conjunction with some other technology. OAuth 2.0 provides the means to secure an API, but it does not provide the API itself. This chapter, *The OAuth Dance*, introduces the OAuth 2.0 protocol, showcasing the authorization code flow used in many web applications today.

Chapter 2 from *OAuth 2 in Action* by Justin Richer and Antonio Sanso

The OAuth Dance

This chapter covers

- An overview of the OAuth 2.0 protocol
- The different components in an OAuth 2.0 system
- How the different components communicate with each other
- What the different components communicate.

By now you have a decent overview of what the OAuth 2.0 protocol is and why it is important. You also likely have an idea of how and where you might want to use the protocol, but what steps do you have to take to make an OAuth transaction? What do you end up with when you're done with an OAuth transaction? How does this design make OAuth secure?

2.1 Overview of the OAuth 2.0 protocol: getting and using tokens

OAuth is a complex security protocol, with components sending pieces of information to each other in a precise balance that's akin to a technological dance. Funda-

mentally, there are two major steps to an OAuth transaction: issuing a token and using a token. The token represents the access that has been delegated to the client and it plays a central role in every part of OAuth 2.0. While the details of each step vary based on several factors, the canonical OAuth transaction consists of the following sequence of events:

1. The Resource Owner indicates to the Client they would like the Client to act on their behalf (for example, load my photos from that service so I can print them)
2. The Client requests authorization from the Resource Owner at the Authorization Server
3. The Resource Owner grants authorization to the Client
4. The Client receives a Token from the Authorization Server
5. The Client presents the Token to the Protected Resource

Different deployments of the OAuth process can handle each of these steps in slightly different ways, often optimizing the process by collapsing several steps into a single action, but the core process remains essentially the same. Next, we will look at the most canonical example of OAuth 2.0.

2.2 Following an OAuth authorization grant in detail

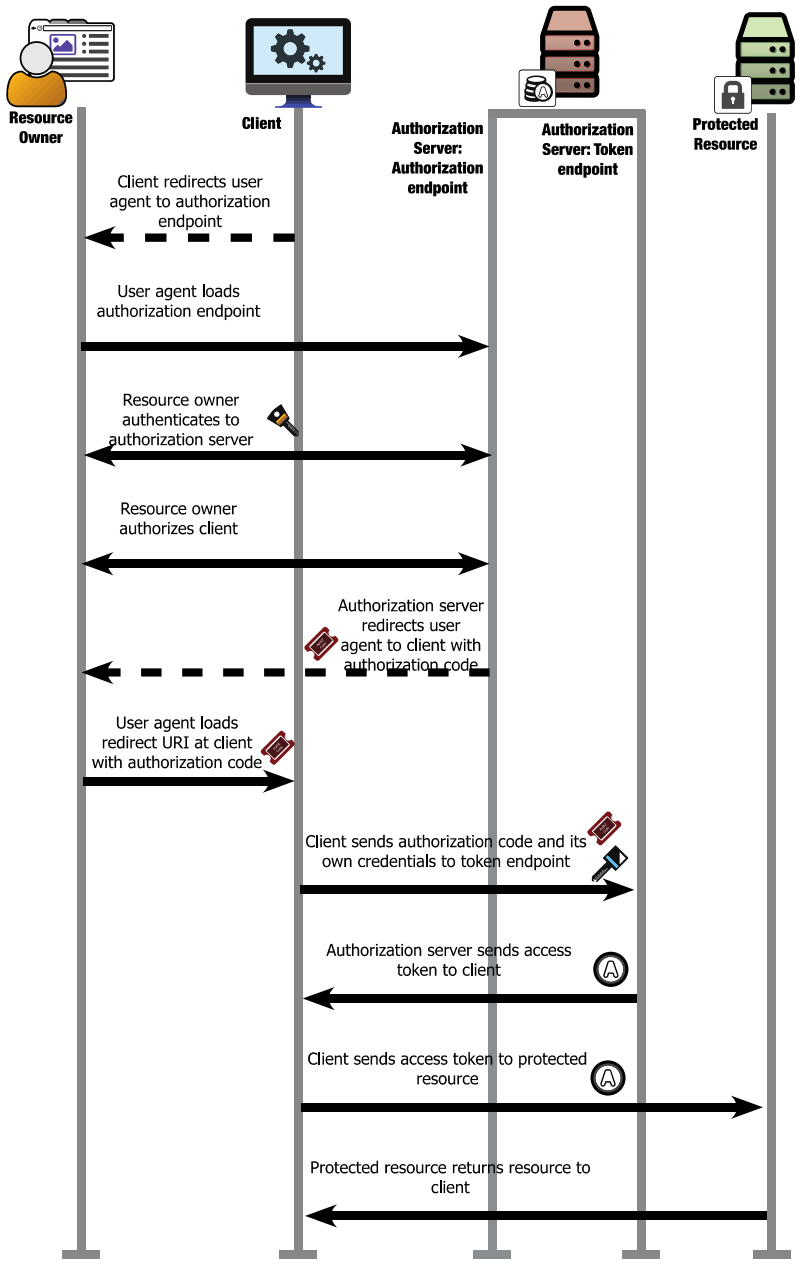
Let us take a look at an OAuth authorization grant process in detail. We are going to be looking at all steps between the actors, tracing the HTTP requests and responses for each step. In particular, we'll follow the authorization code grant used with web-based client applications. These clients will be interactively authorized directly by the resource owner.

> **NOTE**: The examples in this chapter are pulled from the exercise code that we'll use later in the book. While you don't need to understand the exercises to follow what's going on, it might help to run through some of the completed examples in appendix A. Note that the use of *localhost* throughout these examples is purely coincidental, as OAuth can and does work across multiple independent machines.

The *authorization code grant* uses a temporary credential, the authorization code, to represent the resource owner's delegation to the client, and it looks like this.

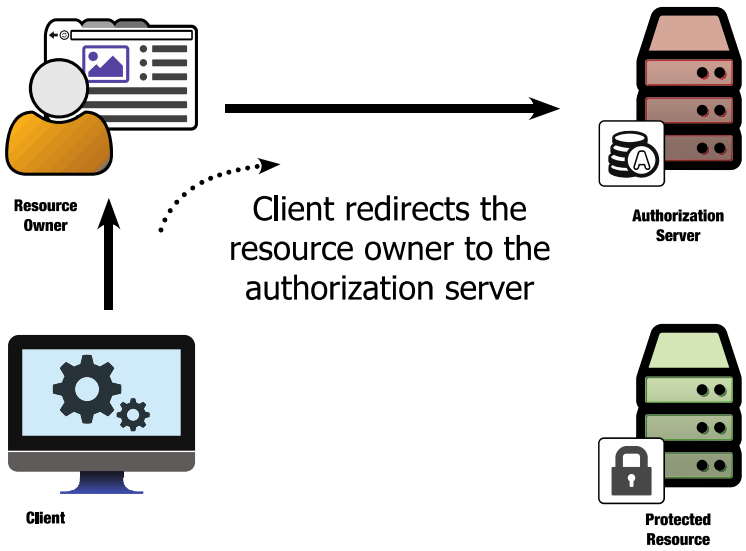
Let me break this down into individual steps. First, the resource owner goes to the client application and indicates to the client they would like it to use a particular protected resource on their behalf. For instance, this is where the user would tell the printing service to use a specific photo storage service.

When the client realizes that it needs to get a new OAuth access token, it sends the resource owner to the authorization server with a request from the client asking to be delegated authority by that resource owner. For example, our photo printer could ask the photo storage service for the ability to *read* the photos stored there.



How do I find the server?

In order to remain maximally flexible, OAuth pushes many details of a real API system out of scope. In particular, the way that the client knows how to talk to a given protected resource, or how the client finds the authorization server tied to that protected resource, are not specified by OAuth. Some protocols built on top of OAuth, like OpenID Connect and UMA, do solve these problems in standard ways, and we will cover those in chapters 13 and 14. For the purpose of demonstrating OAuth itself, we assume the client has been statically configured to know how to talk to both the protected resource and the authorization server.



Since we have a web client, this takes the form of an HTTP redirect to the authorization server's authorization endpoint. The response from the client application looks like this:

```
HTTP/1.1 302 Moved Temporarily
x-powered-by: Express
Location:
     http://localhost:9001/authorize?response_type=code&scope=foo&client_id=o
     auth-client-
     1&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fcallback&state=Lwt50DDQKU
     B8U7jtfLQCVGDL9cnmwHH1
Vary: Accept
Content-Type: text/html; charset=utf-8
Content-Length: 444
Date: Fri, 31 Jul 2015 20:50:19 GMT
Connection: keep-alive
```

This redirect to the browser causes the browser to send an HTTP GET to the authorization server:

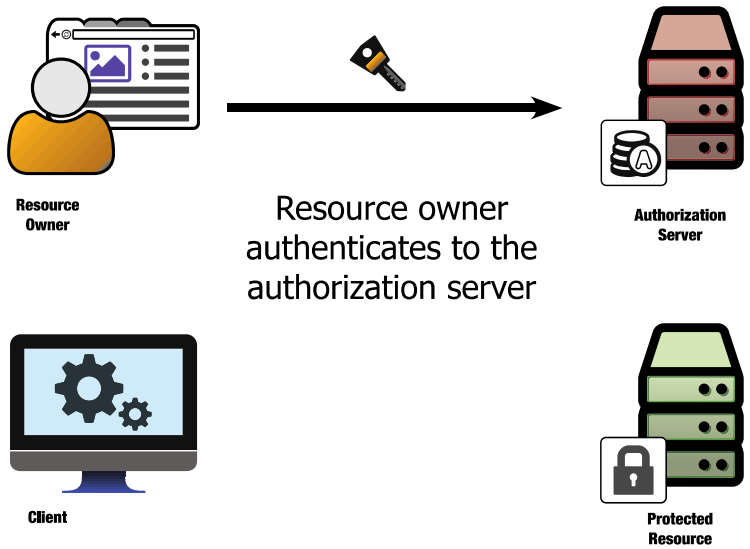
```
GET /authorize?response_type=code&scope=foo&client_id=oauth-client-
    1&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fcallback&state=Lwt50DDQKU
    B8U7jtfLQCVGDL9cnmwHH1 HTTP/1.1
Host: localhost:9001
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:39.0)
    Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost:9000/
Cookie: i18next=en
Connection: keep-alive
```

The client identifies itself and requests particular items, such as scopes, by including query parameters in the URL it sends. The authorization server can parse those parameters and act accordingly, even though the client isn't making the request directly.

Viewing the HTTP transaction

All of the HTTP transcripts were captured using off-the-shelf tools, and there are quite a number of them out there. Browser inspection tools, like the Firebug plugin for Firefox, allow comprehensive monitoring and manipulation of front channel communications. The back channel can be observed using a proxy system or a network packet capture program like Wireshark or Fiddler.

Next, the authorization server will usually require the user to authenticate. This step is essential in determining who the resource owner is and what rights they're allowed to delegate to the client.

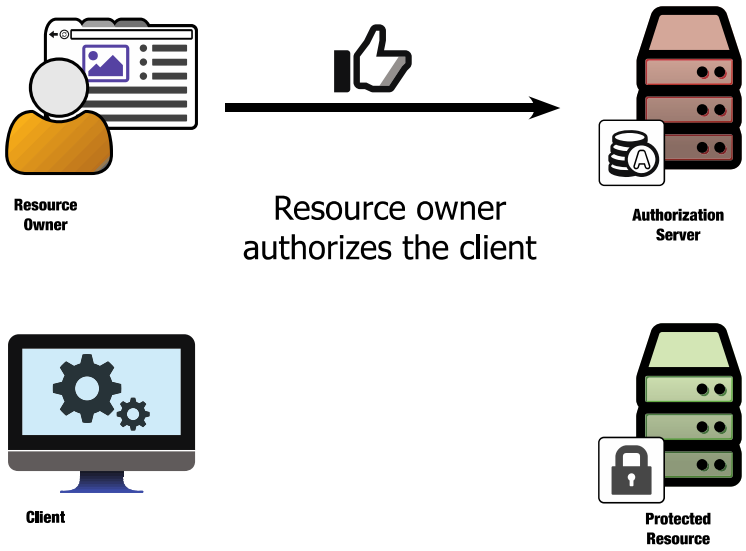


The user's authentication passes directly between the user (and their browser) and the authorization server; it's never seen by the client application. This essential aspect protects the user from having to share their credentials with the client application, the anti-pattern that OAuth was invented to combat (as I discussed in the previous chapter).

Additionally, since the resource owner interacts with the authorization endpoint through a browser, their authentication happens through a browser as well. Thus, a wide variety of authentication techniques are available to the user authentication process. OAuth does not dictate the authentication technology, and the authorization server is free to choose methods such as a username/password pair, cryptographic certificates, security tokens, , or any number of other possibilities. We have to trust the web browser to a certain extent here, especially if the resource owner is using a simple authentication method like username and password, but the OAuth protocol is designed to protect against several major kinds of browser-based attacks, which we will cover in chapters 7, 8, and 9.

This separated approach insulates the client from changes to the user's authentication methods, allowing a simple client application to benefit from emergent techniques, such as risk-based heuristic authentication applied at the authorization server. This does not convey any information about the authenticated user to the client, however; this is a topic we'll cover in depth in chapter 11.

Next, the user authorizes the client application:

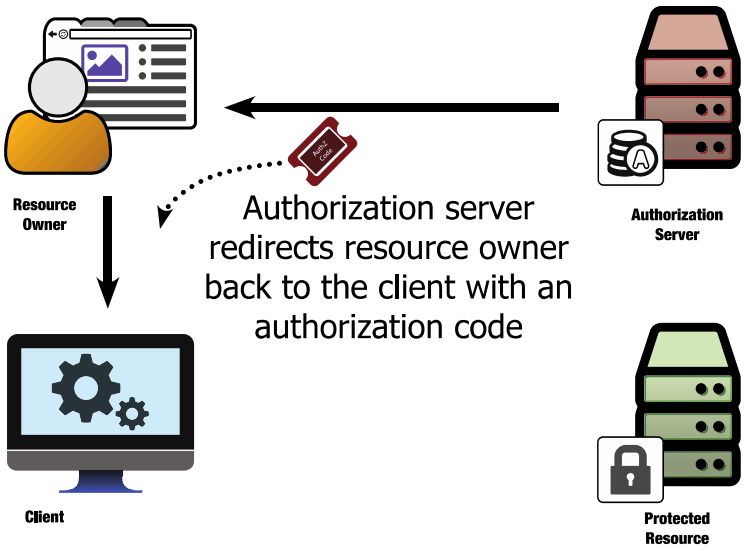


In this step, the resource owner chooses to delegate some portion of their authority to the client application, and the authorization server has many different options for how to make this work. The client's request can include an indication of what kind of access it's looking for (known as the OAuth scope, discussed in section 2.4). The autho-

rization server can allow the user to deny some or all of these scopes, or it can let the user approve or deny the request.

Furthermore, many authorization servers allow the storage of this authorization decision for future use. If this is used, then future requests for the same access by the same client will not prompt the user interactively. The user will still be redirected to the authorization endpoint, and will still need to be logged in, but the decision of whether to delegate authority to the client will have already been made during a previous attempt. The authorization server can even override the end user's decision based on an internal policy such as a client whitelist or blacklist.

Next, the authorization server redirects the user back to the client application:



This takes the form of an HTTP redirect to the client's `redirect_uri`.

```
HTTP 302 Found
Location:
     http://localhost:9000/oauth_callback?code=8V1pr0rJ&state=Lwt50DDQKUB8U7j
     tfLQCVGDL9cnmwHH1
```

This in turn causes the browser to issue the following request back to the client:

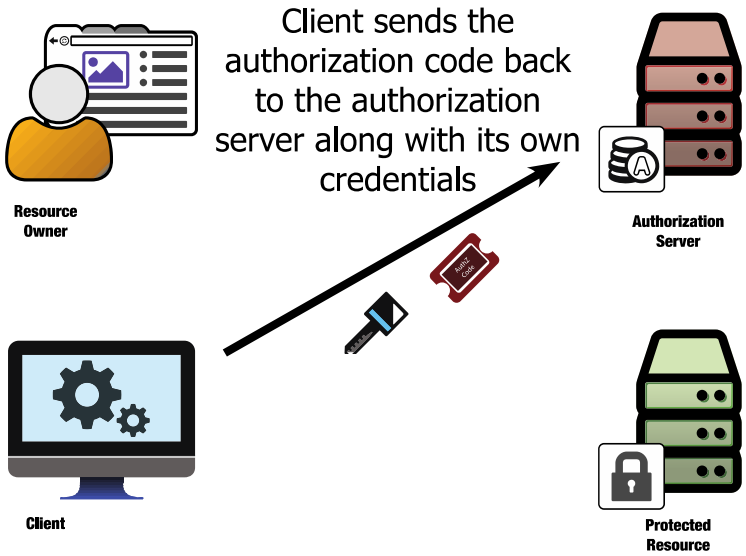
```
GET /callback?code=8V1pr0rJ&state=Lwt50DDQKUB8U7jtfLQCVGDL9cnmwHH1 HTTP/1.1
Host: localhost:9000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:39.0)
     Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
```

Notice that this is on the *client* and not on the *authorization server*

```
Referer:
     http://localhost:9001/authorize?response_type=code&scope=foo&client_id=o
     auth-client-
     1&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fcallback&state=Lwt50DDQKU
     B8U7jtfLQCVGDL9cnmwHH1
Cookie: i18next=en
Connection: keep-alive
```

Since we're using the *authorization code* grant type, this redirect includes the special `code` query parameter. The value of this parameter is a one-time-use credential known as the *authorization code*, and it represents the result of the user's authorization decision. The client can parse this parameter to get the authorization code value when the request comes in, and it will use that code in the next step. The client will also check that the value of the `state` parameter matches the value that it sent in the previous step.

Now that the client has the `code`, it can send it back to the authorization server on its token endpoint:



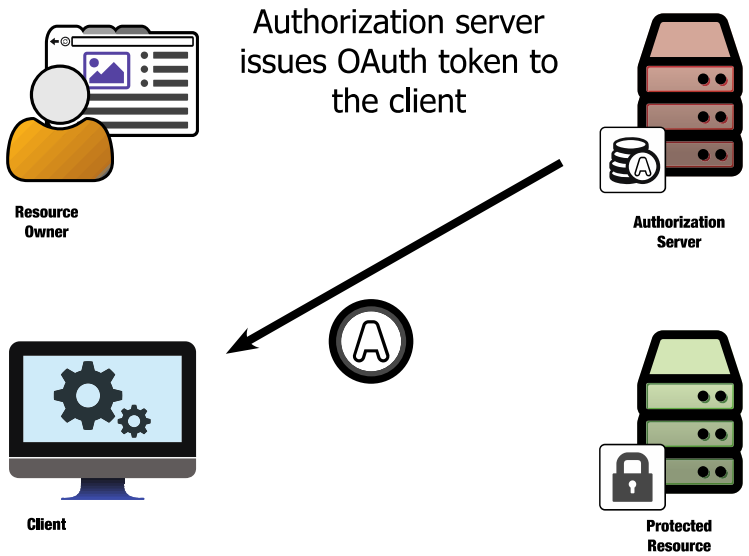
The client performs an HTTP POST with its parameters as a form-encoded HTTP entity body, passing its `client_id` and `client_secret` as an HTTP Basic authorization header. This HTTP request is made directly between the client and the authorization server, without involving the browser or resource owner at all.

```
POST /token
Host: localhost:9001
Accept: application/json
Content-type: application/x-www-form-encoded
Authorization: Basic b2F1dGgtY2xpZW50LTE6b2F1dGgtY2xpZW50LXNlY3J1dC0x

grant_type=authorization_code&
     redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fcallback&code=8V1pr0rJ
```

This separation between different HTTP connections ensures that the client can authenticate itself directly without other components being able to see or manipulate the token request.

The authorization server takes in this request and, if valid, issues a token.



The authorization server performs a number of steps to ensure the request is legitimate. First, it validates the client's credentials (passed in the Authorization header here) to determine which client is requesting access. Then, it reads the value of the `code` parameter from the body and looks up any information it has about that authorization code, including which client made the initial authorization request, which user authorized it, and what it was authorized for. If the authorization code is valid, has not been used previously, and the client making this request is the same as the client that made the original request, the authorization server generates and returns a new access token for the client.

This token is returned in the HTTP response as a JSON object:

```
HTTP 200 OK
Date: Fri, 31 Jul 2015 21:19:03 GMT
Content-type: application/json

{
"access_token": "987tghjkiu6trfghjuytrghj",
"token_type": "Bearer"
}
```

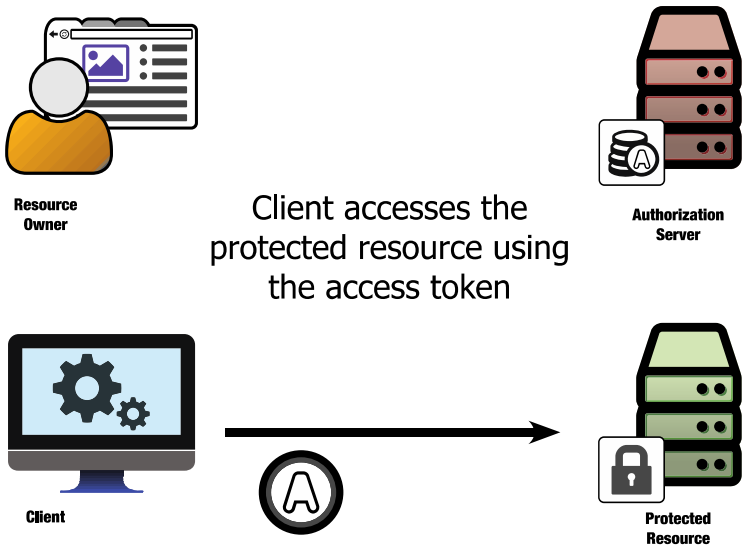
The client can now parse the token response and get the access token value from it to be used at the protected resource. In this case, we have an OAuth Bearer token, as indicated by the `token_type` field in the response. The response can also include a

refresh token (used to get new access tokens without asking for authorization again) as well as additional information about the access token, like a hint as to the token's scopes and expiration time. The client can store this access token in a secure place for as long as it wants to use the token, even after the user has left.

The right to bear tokens

The core OAuth specifications deal with **bearer** tokens, which means that anyone who carries the token has the right to use it. All of our examples will also be using bearer tokens throughout the book, except where specifically noted. Bearer tokens have particular security properties, which are enumerated in chapter 10, and I discuss non-bearer tokens in chapter 15 as well.

With the token in hand, the client can present the token to the protected resource:



The client has several methods for presenting the access token, and in this example we're going to use the recommended method of using the Authorization header:

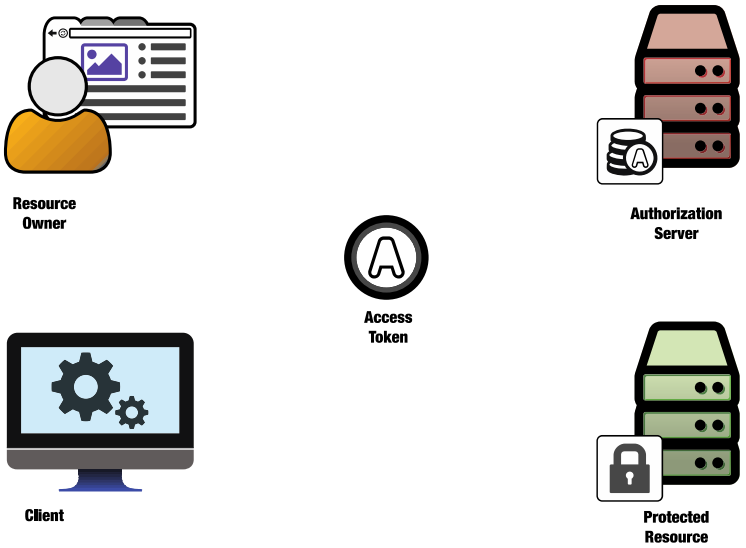
```
GET /resource HTTP/1.1
Host: localhost:9002
Accept: application/json
Connection: keep-alive
Authorization: Bearer 987tghjkiu6trfghjuytrghj
```

The protected resource can parse the token out of the header, determine if it's still valid, look up information regarding who authorized it and what it was authorized for, and return the response accordingly. A protected resource has a number of options for

doing this token lookup, which we'll cover in greater depth in a future chapter. The simplest option is for the resource server and the authorization server to share a database that contains the token information. The authorization server writes new tokens into the store when they're generated, and the resource server reads tokens from the store when they are presented.

2.3 OAuth's actors: clients, authorization servers, resource owners, and protected resources

As we touched on in the last section, there are four main actors in an OAuth system: clients, resource owners, authorization servers, and protected resources. Each of these components is responsible for different parts of the OAuth protocol, and all work together to make the OAuth protocol work.



An OAuth **client** is a piece of software that is attempting to access the protected resource on behalf of the resource owner, and it uses OAuth to get that access. Thanks to the design of the OAuth protocol, the client is generally the simplest component in an OAuth system, and its responsibilities are largely around getting tokens from the authorization server and using tokens at the protected resource. The client does not have to understand the token, nor should it ever need to inspect the token's contents. Instead, the client uses the token as an opaque string. An OAuth client can be a web application, a native application, or even an in-browser JavaScript application, and I cover the differences between these kinds of clients in chapter 6. In our cloud printing example, the printing service is the OAuth client.

An OAuth **protected resource** is available through an HTTP server and it requires an OAuth token to be accessed. The protected resource needs to validate the tokens presented to it and determine whether and how to serve requests. In an OAuth architecture, the protected resource has the final say as to whether or not to honor a token. In our cloud printing example, the photo storage site is the protected resource.

A **resource owner** is the entity with the authority to delegate access to the client. Unlike other parts of the OAuth system, the resource owner is not a piece of software. In most cases, the resource owner is the person using the client software to access something they control. For at least part of the process, the resource owner interacts with the authorization server using a web browser (more generally known as the user agent). The resource owner might also interact with the client using a web browser, as they do in our demonstration, but that's entirely dependent on the nature of the client. In our cloud printing example, the resource owner is the end user who wants to print their photos.

An OAuth **authorization server** is an HTTP server that acts as the central component to an OAuth system. The authorization server authenticates the resource owner and client, provides mechanisms for allowing resource owners to authorize clients, and issues tokens to the client. Some authorization servers also provide additional capabilities such as token introspection and remembering authorization decisions. In our cloud printing example, the photo storage site runs its own in-house authorization server for its protected resources.

2.4 OAuth Components: Tokens, scopes, and authorization grants

In addition to these actors, the OAuth ecosystem depends on several other mechanisms, both conceptual and physical. These are the bits that connect the actors above in a larger protocol.

2.4.1 Access tokens

An OAuth **access token**, sometimes known as a token, is an artifact issued by the authorization server to a client that indicates the rights the client has been delegated. OAuth does not define a format or content for the token itself, but it always represents the combination of the client's requested access, the resource owner that authorized the client, and the rights conferred during that authorization (usually including some indication of the protected resource).

OAuth tokens are opaque to the client, which means that the client has no need (and often no ability) to look at the token itself. The client's job is to carry the token, requesting it from the authorization server and presenting it to the protected resource. The token is not opaque to everybody in the system: the authorization server's job is to issue the token, and the protected resource's job is to validate the token. As such, they both need to be able to understand the token itself and what it stands for. However, the client is completely oblivious to all of this. This approach allows the client to be much simpler than it would otherwise need to be, as well as giving the authorization server and protected resource incredible flexibility in how these tokens are deployed.

2.4.2 Scopes

An OAuth **scope** is a representation of a set of rights at a protected resource. Scopes are represented by strings in the OAuth protocol, and they can be combined into a set by using a space-separated list. As such, the scope value cannot contain the space character. The format and structure of the scope value is otherwise undefined by OAuth.

Scopes are an important mechanism for limiting the access granted to a client. Scopes are defined by the protected resource, based on the API that it is offering. Clients can request certain scopes, and the authorization server can allow the resource owner to grant or deny particular scopes to a given client during its request. Scopes are generally additive in nature.

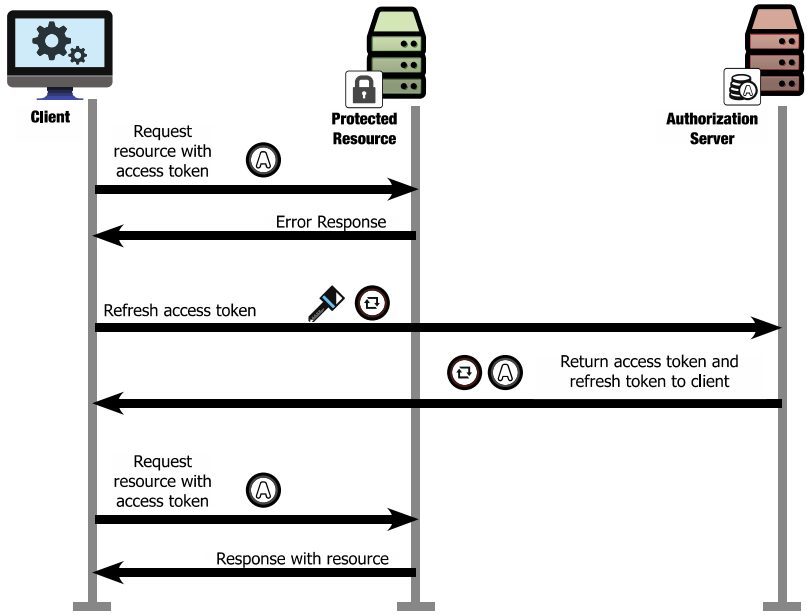
Going back to our cloud printing example from before. The photo storage service's API defines several different scopes for accessing the photos: read-photo, read-metadata, update-photo, update-metadata, create, and delete. The photo printing service only needs to be able to read the photos in order to do its job, and so it asks for the read-photo scope. Once it has an access token with this scope, the printer is able to read photos and print things out as requested. If the user decides to use an advanced function that prints a series of photographs into a book based on their date, the printing service will need the additional read-metadata scope. Since this is an additional access, the printing service needs to ask the user to authorize them for this additional scope using the regular OAuth process. Once the printing service has an access token with both scopes, it can perform actions that require either of them, or both of them together, using the same access token.

2.4.3 Refresh tokens

An OAuth **refresh token** is similar in concept to the access token, in that it is issued to the client by the authorization server and the client does not know, or care, what is inside the token. What is different is that the token is not ever sent to the protected resource. Instead, the client uses the refresh token to request new access tokens without involving the resource owner.

Why would a client need to bother with a refresh token? In OAuth, an access token could stop working for a client at any point. The user could have revoked the token, it could have expired, or some other system trigger might have made the token invalid. The client will usually find out about the token being invalid by using it and getting an error response. The client could get the resource owner to authorize them again. But what if the resource owner's not there anymore?

In OAuth 1.0, the client had no recourse but to wait for the resource owner's return. To avoid this, tokens in OAuth 1.0 tended to live forever until explicitly revoked. This is a bit problematic as it increases the attack surface for a stolen token: the attacker can keep using the stolen token forever. In OAuth 2.0, access tokens were given the option to automatically expire, but we needed a way to access resources when the user wasn't there anymore. The refresh token now takes the place of the long-lived token, but instead of being used to get resources, it is used to get new access tokens



which, in turn, can get the resources. This limits the exposure of the refresh token and the access token in separate but complimentary ways.

Refresh tokens also give the client the ability to down-scope their access. If a client is granted scopes `A`, `B`, and `C`, but it knows that it only needs scope `A` to make a particular call, it can use the refresh token to request an access token for only scope `A`. This lets a smart client follow the security principle of least privilege without burdening less-smart clients with trying to figure out what privileges an API needs. Years of deployment experience has shown us that OAuth clients tend to be anything but smart, but it's still good to have the advanced capability for those who want to exercise it.

What if the refresh token itself doesn't work? The client can always bother the resource owner again, when they are available. In other words, the fallback state for an OAuth client is to do OAuth again.

2.4.4 Authorization grants

An **authorization grant** is the means by which an OAuth client is given access to a protected resource using the OAuth protocol, and if successful it ultimately results in the client getting a token. This is likely one of the most confusing terms in OAuth 2.0, since the term is used to define both the specific mechanism by which the user delegates authority, as well as the act of delegation itself. This is further confused by the authorization code grant type, which we laid out in detail above, as developers will sometimes look at the authorization code that's passed back to the client and mistakenly assume that this artifact – and this artifact alone – is the authorization grant.

While it's true that the authorization code represents a user's authorization decision, it's not itself an authorization grant. Instead, the entire OAuth process is the authorization grant: the client sending the user to the authorization endpoint, receiving the code, and finally trading the code for the token.

In other words, the authorization grant is the method for getting a token. In this book, as in the OAuth community as a whole, I occasionally refer to this as a **flow** of the OAuth protocol. There are several different kinds of authorization grants in OAuth, each with its own characteristics. I'll cover these in detail in chapter 6, but most of our examples and exercises, such as the ones in the previous section, are going to be using the authorization code authorization grant type.

2.5 Interactions between OAuth's actors and components: back channel, front channel, and endpoints

Now that we know the different parts of an OAuth system, take a look at how exactly they communicate with each other. OAuth is an HTTP-based protocol, but unlike most HTTP-based protocols, OAuth communication doesn't always happen through a simple HTTP request and response.

OAuth over non-HTTP channels

While OAuth is defined only in terms of HTTP, several specifications have defined how to move different parts of the OAuth process to non-HTTP protocols. For instance, there are draft standards defining how to use OAuth tokens over GSS-API[a] and CoAP[b]. All of these still use HTTP to bootstrap the process, and they tend to translate the HTTP-based OAuth components as directly as possible to these other protocols.

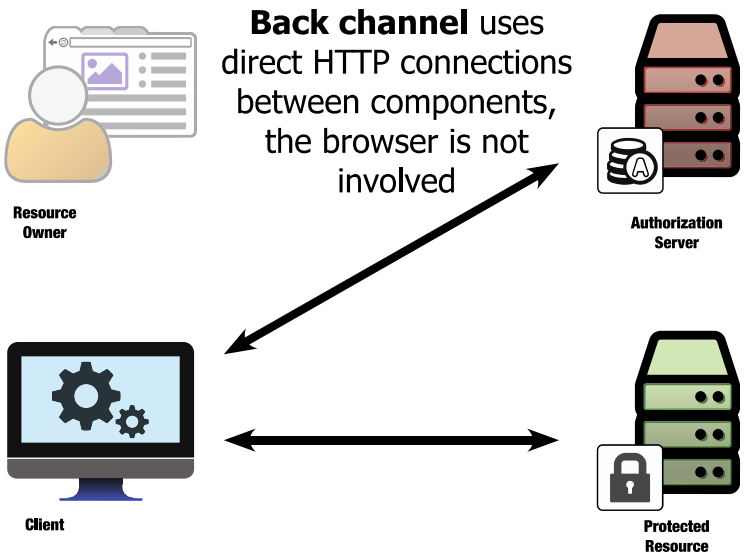
[a] RFC 7628
[b] https://tools.ietf.org/html/draft-ietf-ace-oauth-authz

2.5.1 Back-channel Communication

Many parts of the OAuth process use a normal HTTP request and response format to communicate to each other. Since these requests generally happen outside the purview of the resource owner and user agent, they're collectively referred to as back-channel communication.

These requests and responses make use of all the regular HTTP mechanisms to communicate: headers, query parameters, methods, and entity bodies can all contain information vital to the transaction. Note that this might be a bit more of the HTTP stack than you're used to, as many simple web APIs allow the client developer to pay attention to the response body.

The authorization server provides a token endpoint that the client uses to request access tokens and refresh tokens. The client calls this endpoint directly, presenting a form-encoded set of parameters that the authorization server parses and processes. The authorization server then responds with a JSON object representing the token.



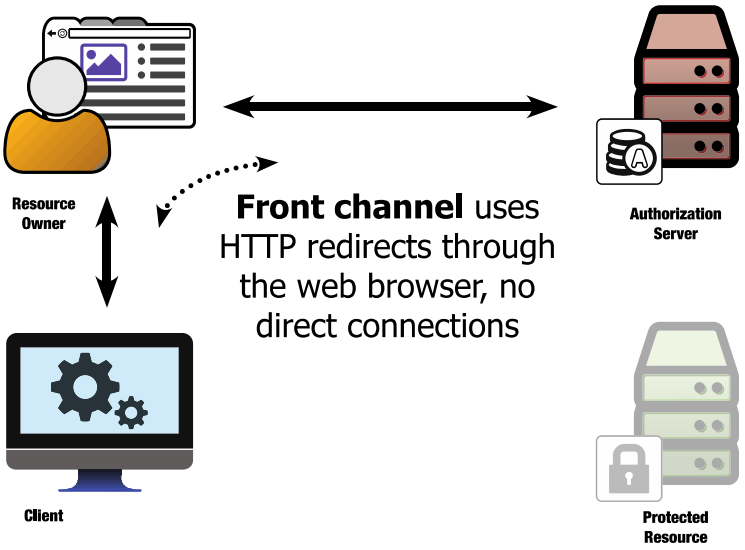
Additionally, when the client connects to the protected resource, it's also making a direct HTTP call in the back channel. The details of this call are entirely dependent on the protected resource, as OAuth can be used to protect an extraordinarily wide variety of APIs and architecture styles. In all of these, the OAuth token is presented by the client and the protected resource must be able to understand the token and the rights that it represents.

2.5.2 Front-channel Communication

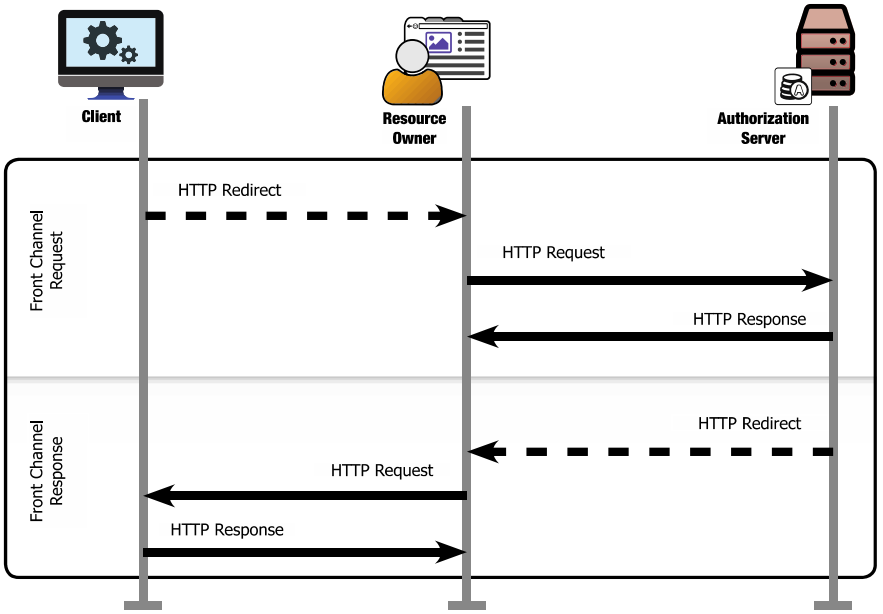
In normal HTTP communication, as we saw, the HTTP client sends a request that contains headers, query parameters, an entity body, and other pieces of information directly to a server. The server can then look at those pieces of information and figure out how to respond to the request, using an HTTP response containing headers, an entity body, and other pieces of information. However, in OAuth, there are several instances where two components cannot make direct requests and responses of each other, such as when the client interacts with the authorization endpoint of the authorization server. Front-channel communication is a method of using HTTP requests to communicate indirectly between two systems through an intermediary web browser.

This technique isolates the sessions on either side of the browser, which allows it to work across different security domains. For instance, if the user needs to authenticate to one of the components, they can do so without exposing their credentials to the other system. We can keep information separate and still communicate in the presence of the user.

How can two pieces of software communicate without ever talking to each other? Front channel communication works by attaching parameters to a URL and indicating that the browser should follow that URL. The receiving party can then parse the



incoming URL, as fetched by the browser, and consume the presented information. The receiving party can then respond by redirecting the browser back to a URL hosted by the originator, using the same method of adding parameters. The two parties are thus communicating with each other indirectly through the use of the web browser as an intermediary. This means that each front channel request and response is actually a pair of HTTP request and response transactions.



For example, in the authorization code grant example above, the client needs to send the user to the authorization endpoint, but it also needs to communicate certain parts of its request to the authorization server. To do this, the client sends an HTTP redirect to the browser. The target of this redirect is the server's URL with certain fields attached to it as query parameters:

```
HTTP 302 Found
Location: http://localhost:9001/authorize?client_id=oauth-client-
    1&response_type=code&state=843hi43824h42tj
```

The authorization server can parse the incoming URL, as any other HTTP request, and find the information sent from the client in these parameters. When it's time to return the authorization code to the client, the authorization server sends an HTTP redirect to the browser as well, but this time with the client's `redirect_uri` as the base. The authorization server also includes its own query parameters in the redirect:

```
HTTP 302 Found
Location: http://localhost:9000/oauth_callback?code=23ASKBWe4
```

When the browser follows this redirect, it'll be served by the client application, in this case through an HTTP request. The client can parse the URL parameters from the incoming request. In this way, the client and authorization server can pass messages back and forth to each other through an intermediary without ever talking to each other directly.

What if my client isn't a web app?

OAuth can be used by both web applications and native applications, but both need to use the same front channel mechanism to get information back from the authorization endpoint. The front channel always uses a web browser and HTTP redirects, but they don't always have to be served by a regular web server in the end. Fortunately, there are a few useful tricks, like internal webservers, application-specific URI schemes, and push notifications from a backend service that can be used. As long as the browser can invoke a call on that URI, it will work. We'll explore all of these options in detail in chapter 6.

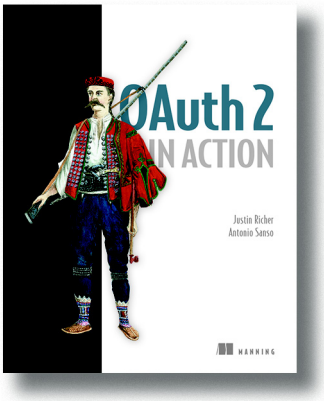
All information that's passed through the front channel is accessible to the browser, both to be read and potentially manipulated before the ultimate request is made. The OAuth protocol accounts for this by limiting the kinds of information that are passed through the front channel, and by making sure that none of the pieces of information used in the front channel can be used on its own to accomplish the task of delegation. In the canonical case we saw in this chapter, some protocols, such as OpenID Connect, offer increased security through mechanisms for these front channel messages to be signed by the client or authorization server to add a further layer of protection, and we'll look at that briefly in chapter 13.

2.6 Summary

OAuth is a protocol with many moving pieces, but it is built of simple actions that add up to a secure method for authorization delegation.

- OAuth is about **getting tokens** and **using tokens**
- Different components in the OAuth system care about different parts of the process
- Components use direct (back channel) and indirect (front channel) HTTP to communicate with each other.

Now that you have learned what OAuth is and how it works, let's start building things! In the next chapter, we will build an OAuth client from scratch.



Think of OAuth 2 like the web version of a valet key. This HTTP-based security protocol allows the users of a service to enable applications to use that service on their behalf without handing over full control. Web and mobile apps can securely access information from other servers for these users, enabling you to give your users functionality and services from other sites. Instead of unsafe password-sharing, OAuth offers a much more secure delegation protocol. OAuth is used everywhere, from large providers like Facebook and Google, to small APIs at startups, and even cloud services, it's the worldwide standard. OAuth 2 is the must-know security protocol on the web today.

OAuth 2 in Action teaches you practical use and deployment of this protocol from the perspective of a client, authorization server, and resource server. You'll begin with an overview of OAuth and a look at its components and interactions. Then, using lots of hands-on examples, you'll build your first OAuth client, followed by an authorization server, and then a protected resource. The second part of the book dives into crucial implementation vulnerability topics. Then you learn about tokens, dynamic client registration, and more advanced topics. This book teaches you to how to distinguish between different OAuth options and choose the right set for your application. By the end of this book, you'll be able to build and deploy applications that use OAuth on both the client and server sides.

What's inside

- Understand OAuth 2 protocol and design
- Authorization with OAuth 2
- Implementation risks
- Building an OAuth 2 environment
- Protecting and accessing REST APIs

Readers need basic programming skills and knowledge of HTTP and JSON.

Working with Web APIs

Good API design balances the technical details of the implementation with the business value desired by the product managers. Good APIs consider security even in the design phase. This chapter gives you an overview of web-based APIs and general API design concerns.

Chapter 2 from *Irresistible APIs: Designing web APIs that developers will love* by Kirsten K. Hunter

Working with web APIs

This chapter covers

- Structure of a simple API
- Ways to inspect calls to an API
- Interaction between an API and a client application
- Deployment of the sample API and application on your system

In the next few chapters, I'll cover the server-client interaction in detail, but in this chapter I'll help you understand the concepts with a simple example of an API and sample application. Most basic API examples use a to-do list but that's kind of overused, so I decided to go a different way: I've selected a list application with pizza toppings. Note that this particular application is simple by design; the goal is to show you how to interact with the API, and how an application interacts with an API. Of course, if this were a production application it would have a full pizza, or pizzas, and the database wouldn't be shared. But for the goals here, I've taken out as much complexity as possible to make the basic principles clear.

Looking at an API is interesting, but it doesn't necessarily help you to understand how it can drive an application. Additionally, it's challenging to perform

actions such as create and delete in a browser, so in addition to the API I've included a simple application using this API with JavaScript. This application exercises all of the functionality in the API so you can see how an application interacts with a web API.

To get an idea of how this works in practice, I've created a basic API using Node.js, a JavaScript-based web server framework. (You can learn more about this framework at www.nodejs.org.) The API supports all of the needed actions to represent a complete system: create, read, update, and delete. The first task will be to explore the API in a browser using the read functionality.

This application runs on a web host at www.irresistibleapis.com/demo. You can check out the application there and follow along with the concepts in this chapter. If you're a developer and want to explore the code more intimately, use the exercises at the end of the chapter to get the example running on your own system, including both the Node.js application and the HTML/JavaScript web application. In the section 2.6, I also describe the various moving parts to this API and application so you can play with it as you like.

2.1 HTTP basics

To understand the transactions between the client and the server in API interactions, you'll need a basic grasp of how HTTP works. This topic will be covered in more detail in chapter 4, but for now I'll give you some high-level information about the protocol.

HTTP stands for HyperText Transfer Protocol, and you're probably most familiar with it as the way web browsers get information from web servers. An HTTP transaction is composed of a request from the client to the server (like a browser asking for a web page), and a response from the server back to the client (the web page from the server, for a browser). First, I'll describe the elements in an HTTP request. You're familiar with the URL, the address that you type into the address box on a browser, but that address is only one portion of the information sent from your browser to the server in order to process a web request.

2.1.1 HTTP request

Figure 2.1 demonstrates the elements that make up an HTTP request, along with examples of how these sections are used. The HTTP request is usually sent with headers, setting the context for the transaction. An HTTP request always has a method; methods are the verbs of the HTTP protocol. To understand what your browser does, imagine that you're visiting my main website. Here are the pieces of the request that are sent by your browser:

- *Headers: Accept: text/html*—This tells the server that the browser wants to get an HTML-formatted page back. It's the most readable format for humans, so it makes sense that your browser would request it.
- *Method:* `GET`—This is the read method in HTTP and is generally the method used by browsers when reading web pages.

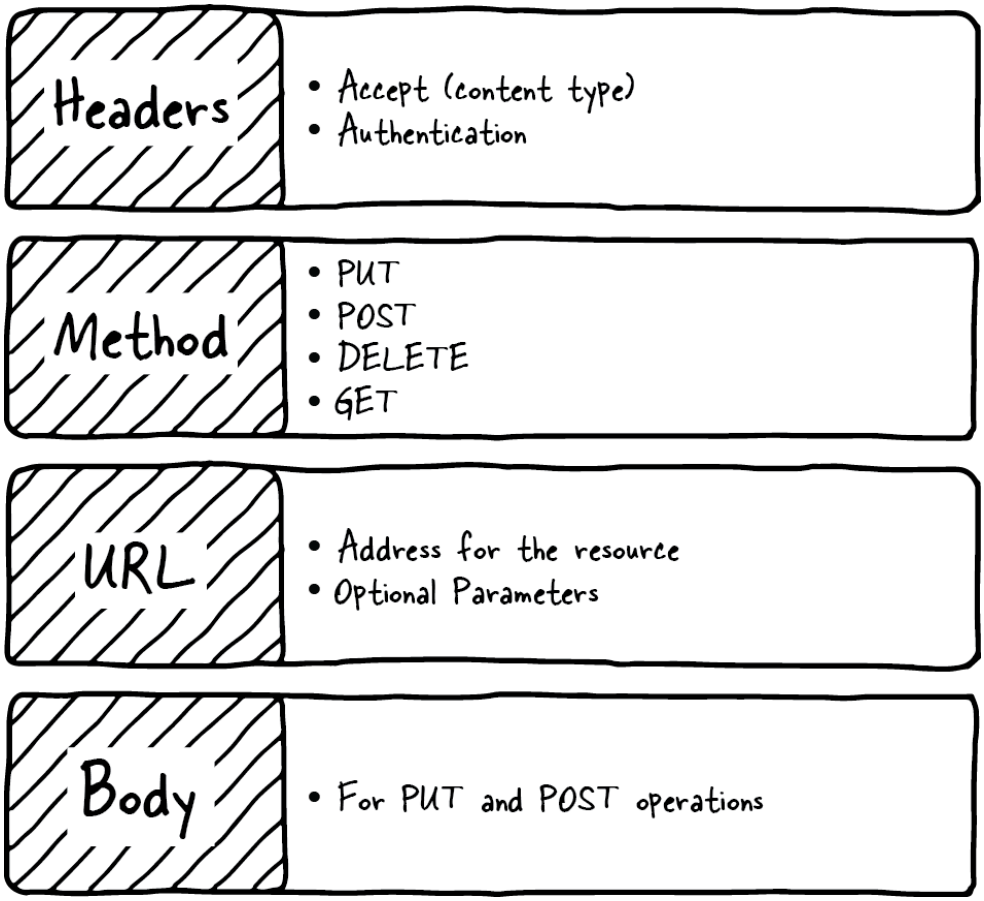


Figure 2.1 An HTTP request will always have a method and will be sent to a specific URL, or resource. Depending on the specific call, headers may be sent to specify information about the request. If the call is designed to write new information to the system, a body will be sent to convey that information.

- *URL: http://irresistibleapis.com*—This is the only piece you actually indicated for the browser.
- *Body: none*—A GET request doesn't need a body, because you're not changing anything on the server—you're just reading what's there.

All of the actions of CRUD (create, read, update, and delete) are represented by methods within HTTP:

- Create: `POST`
- Read: `GET`
- Update: `PUT`
- Delete: `DELETE`

The URL is the unique identifier for the resource. It's just like any other URL on the internet, except in this case it's used to describe the resource in an application system. If parameters are needed for the request, such as a keyword for search, they're included in the parameters of the request. To see how parameters would look, here's an example search request:

```
http://www.example.com/api/v1.0/search?keyword=flintstone&sort=alphabetical
```

In this example, the resource being called is http://www.example.com/api/v1.0/search. The question mark and everything following it are parameters giving more information about what the client wants in the response. A body section is only sent for create (`POST`) and update (`PUT`) transactions.

Next, I'll describe the sections of an HTTP response.

2.1.2 HTTP response

Figure 2.2 shows the elements of a typical HTTP server response. The server is likely to send back several headers giving information about the system and the response. Just as all requests have a method, all responses have a status code. These status codes will be described in more detail in chapter 4, but for now it's sufficient to know that 2XX means that the request was successful, 3XX is a redirect to another location, 4XX is an error in the request from the client, and 5XX means the server had a problem. In the earlier example, calling my website, the server would've responded with the following:

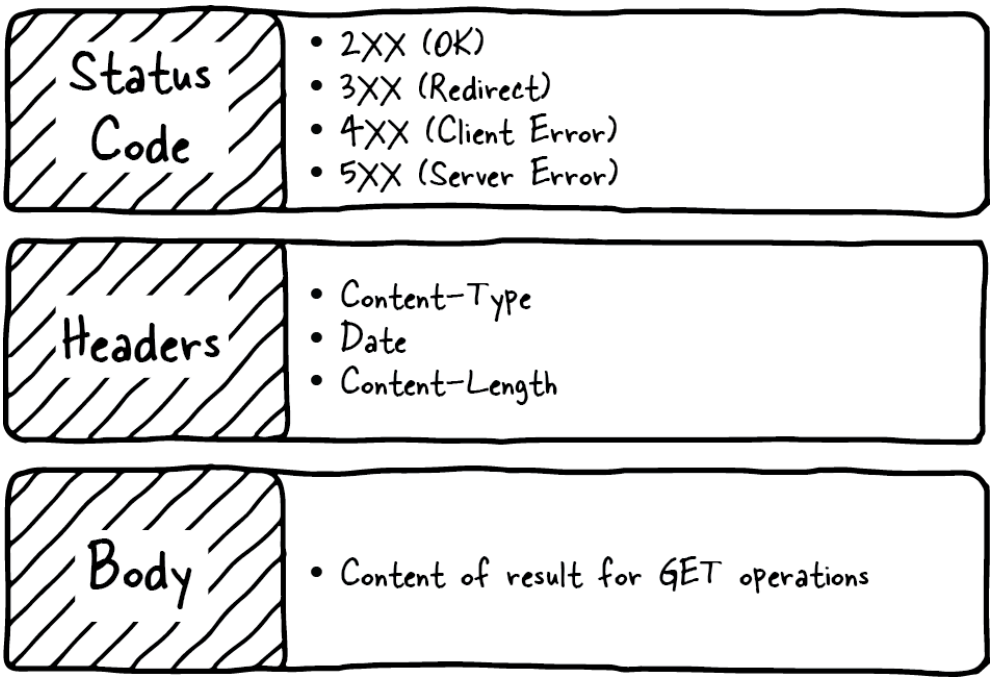


Figure 2.2 A response will always have a status code, and a well-designed platform will send headers to provide information about the response (such as size or the content type). For most requests, a body will be sent back from the server to provide information about the current status of the resource.

- *Status code: 200*—Everything worked correctly.
- *Headers:*
  - *Content-Type: text/html*—as requested by the client
  - *Date: <date of response>*
  - *Content-Length: <size of response>*
- *Body*—The content of the page. This is what you see if you "view source" within the browser—the HTML page that tells the browser how to render the page and what content to display.

2.1.3 HTTP interactions

Every HTTP transaction between a client and server will be composed of a request, sent from the client to the server, and a response, sent from the server back to the client. There's no higher level interaction; each request/response is stateless and starts again from scratch. To help you understand this better, I'll move on to a discussion of a specific API.

2.2 The Toppings API

Many different styles of API are available, but the one I'm going to be using and talking most about here is a Representational State Transfer (REST)-style API, the most common type of web API.

As discussed in chapter 1, REST APIs are designed to work with each resource as a noun. A specific resource within a system has a unique identifier, which is a URL, just like the ones you visit in the browser. This URL identifies the resource in the system and is designed to be understandable when viewed. For example, with a REST API you could view the list of existing toppings with the following request:

```
http://irresistibleapis.com/api/v1.0/toppings
```

These are the actual URLs, retrieved with a `GET` (read) operation. If you put this in a browser, you'll see the results displayed in figure 2.3.

You can visit this URL in your browser right now and get the information about a single topping or a list of toppings. Figure 2.3 shows what this call will look like in a web browser. Go ahead and try both of these calls in your own web browser to see how easy it is to retrieve information from this kind of service. Again, this is just like any other web request, only formatted for a computer to work with.

Now, to view a single topping, you'd take the id field from the list you just retrieved and append it to the URL. Basically, you're saying "Give me the toppings list" and then "but just the one with the id of 1." Almost all APIs work in this way. The parent level is a list of items, and adding an ID will retrieve a single member of the list.

```
http://irresistibleapis.com/api/v1.0/toppings/1
```

The same resource is accessed to update, view. or delete a particular item, simply using different HTTP methods as described in section 2.1 to tell the server what you

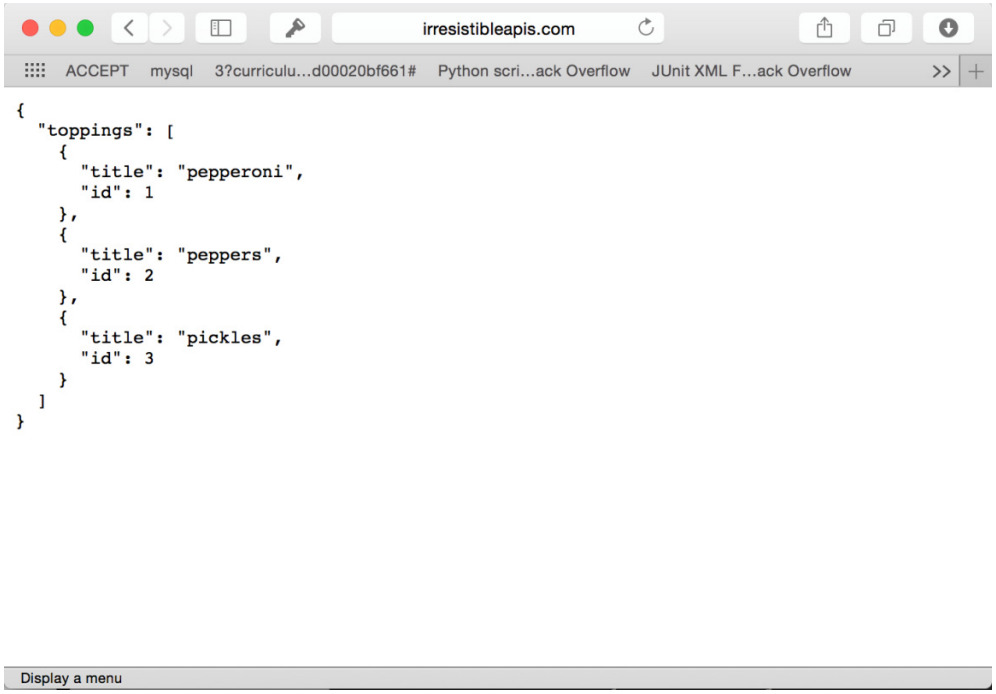


Figure 2.3 Example result of a web call in a browser. The response is JSON, a common markup language for web APIs. As you can see, the formatting makes it easy to understand the content of the response.

want to do. You can add new items by sending a `POST` to the list itself (so in the earlier case, the `/toppings` endpoint would be used to add a new topping). This type of API encourages engagement and innovation by the developers, and consistency across multiple API providers makes it easier to get up and going writing clients.

2.3 Designing the API

To go through the steps, imagine an online website for a pizza parlor. Users are having trouble interfacing with the pizza ordering system and want to be able to customize their pizzas. The company wants to increase customer satisfaction. This represents the Business Value for this platform. Figure 2.4 illustrates each call to the system and how it would be formatted.

To provide this, they need to create a system that consistently allows their users to pick different pizza toppings and keep them in a list (Use Case). The company decides to measure success by determining the increase in people finishing up started orders (Measurements). Fortunately for this example, it's relatively easy to figure out how an API can meet these needs.

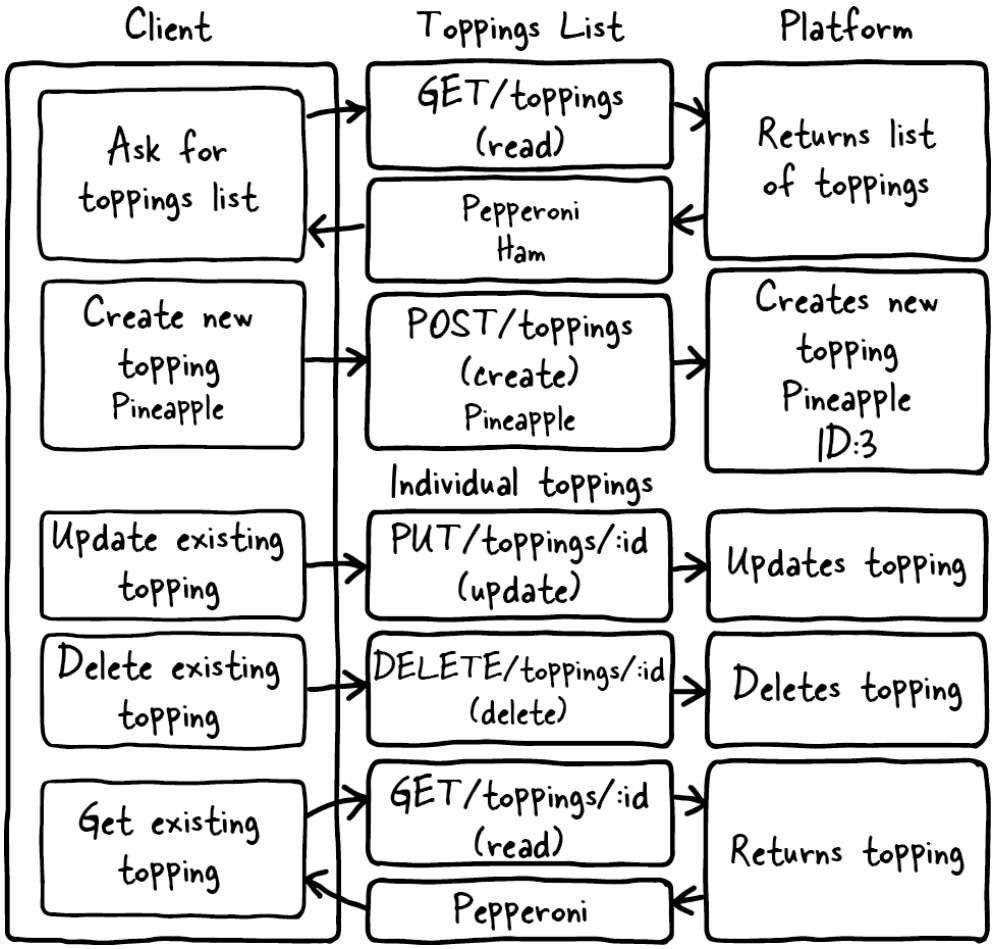


Figure 2.4 This diagram represents the complete set of interactions with the API system. The `GET` request reads the current value of the resource, whether it's a list or an individual item. `POST` is only allowed at the list level, and creates a new resource in the system. `PUT` updates and `DELETE` deletes an existing resource. All four of the needed methods, `Create`, `Read`, `Update`, and Delete method `Delete`, are represented in this diagram.

Because I'm creating a resource-based API, each request will be a unique URL describing one piece of the back-end structure with a method describing what the client wants to do with that resource. In this case, I have only two different types of resources: individual toppings and lists of toppings. Individual topping resources such as `/api/v1.0/toppings/1` are used for viewing, editing, and deleting a single topping. The list resource `/api/v1.0/toppings` is used for viewing all toppings or for adding a new topping to the list. Table 2.1 shows each call to the API and a description of what it does.

Table 2.1 API calls

API Call	Description
`GET /api/v1.0/toppings`	List current toppings
`GET /api/v1.0/toppings/1`	View a single topping
`POST /api/v1.0/toppings`	Create a new topping
`PUT /api/v1.0/toppings/1`	Update an existing topping
`DELETE /api/v1.0/toppings/1`	Delete an existing topping

And that's it. The platform features create, read, update, and delete operations available to you by combining the HTTP methods with the URLs for your resources. But what do you get when you make these calls? When you `GET` the resource for a single topping, you get information just about that topping. Try this now in your browser: http://irresistibleapis.com/api/v1.0/toppings/1.

Listing 2.1 Retrieving a single topping

```
GET /api/v1.0/toppings/1
{
  "topping": {
    "id": 1,
    "title": "Pepperoni"
  }
}
```

Curly braces indicate an object

This response is represented in JavaScript Object Notation (JSON), a formatting syntax first described in chapter 1. JSON will be covered in more detail in chapter 4, but for now you can see how the data is structured. If you want more information about JSON you can find it at http://json.org. The curly braces indicate an object, which is a group of pairings of names and values. What's represented here is a JSON structure describing a single object—a "topping," which has an ID of 1 and a title of Pepperoni. This is the same resource address a client can access to view, delete, or update an existing topping. This means that the URL for the single topping is actually the toppings list of http://irresistibleapis.com/api/v1.0/toppings followed by the ID of the topping from within this structure—so http://irresistibleapis.com/api/v1.0/toppings/1.

If you `GET` the resource for the list of toppings directly, the returned information includes a list instead of a single object. Call this URL in your browser to see the list: http://irresistibleapis.com/api/v1.0/toppings.

Listing 2.2 Retrieving a list of all toppings

```
GET /api/v1.0/toppings
{
  "toppings": [
    {
      "id": 1,
```

Curly braces indicate dictionaries

Square braces indicate lists

```
      "title": "Pepperoni"
    },
    {
      "id": 2,
      "title": "Pineapple"
    }
  ]
}
```

In this case, because the request was for a list of objects, square brackets demonstrate that the returned object contains a *list* of "toppings." Each individual topping looks the same as listing 2.1. Again, this is simply how information is represented in JSON. To understand these calls and responses, just remember that an object (with keys and values) is represented by curly braces, and a list (an unnamed collection of items) is represented with square brackets. In some programming languages these are referred to as hashes and arrays.

Both of these calls can be made from a standard web browser. If other people have added items to the list, you'll see those included in the list view as well; this is a live call into the API system and will return the appropriate information. In this case, the API is generated by node. If you're a developer who's interested in learning more about the back end of the system, the advanced exercise at the end of the chapter will give you information about how to run this system on your own, as well as the application running on top of the API.

This simple API interaction gives you the opportunity to start understanding some of the topics covered in chapter 4.

2.4 Using a web API

You can interact with this API in various ways, as you'll learn in this section. Feel free to try any or all of these approaches to see how the interaction works.

2.4.1 Browser

A browser can make `GET` calls to specific resources very easily. Note that this is easy in the case of my demo API because there's no authentication to worry about. The challenge is that the browser doesn't have any way to easily update, delete, or create new items. Using the developer tools or web inspector in your browser can give you more information about the call as well.

For instance, the Chrome web browser has developer tools that allow you to inspect the traffic it's processing. Figure 2.5 shows what these tools look like in the browser. I'll break down what you're seeing here in terms of what I described earlier. Note that the Chrome tools are showing the request and response combined together in this tab.

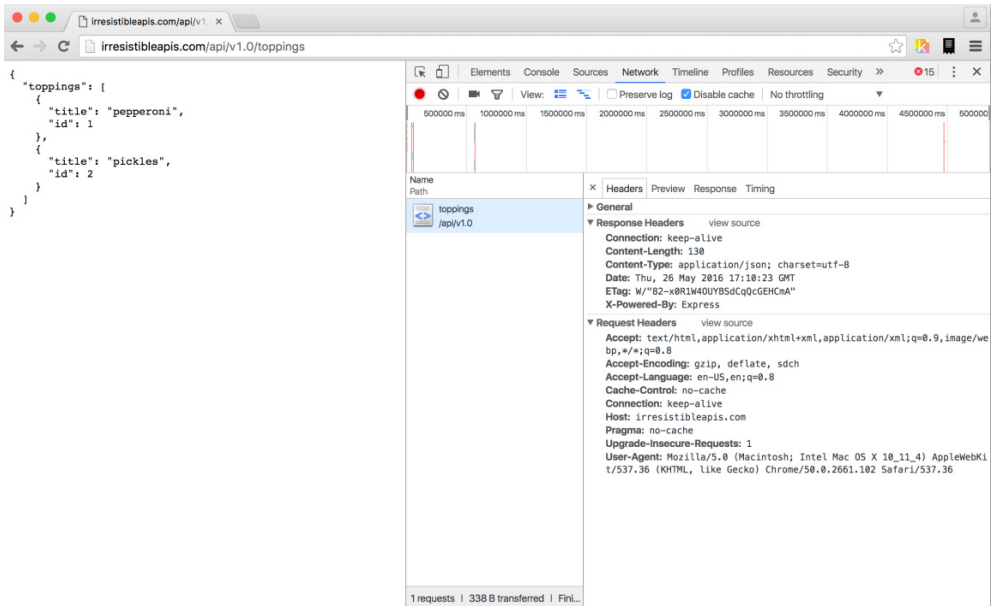


Figure 2.5 The Chrome browser makes it possible to see information about the request and response headers, the body of the request or response, and other useful information about the transaction. Although browsers aren't designed to send `PUT` or `DELETE` responses, the information provided here can go a long way in helping you to understand the interactions with the platform.

For the request:

- Headers:
  - *Accept: text/html, application/xhtml+xml,application/xml; q=0.9,image/webp; */*; q=0.8*—This is the list of accepted formats for this browser request, in order of preference. Because it includes "*/*" – or "any content type" late in the list, the browser will accept any type of response and do the best it can with it.
  - Many other headers are shown in figure 2.5. Take a look at them and run the same request on your system to see how they change and what stays the same in each request/response transaction.
- *Method*—`GET`
- *URL*—http://irresistibleapis.com/api/v1.0/toppings
- *Request Body*—none
- *Status code*—200 OK

2.4.2 Command line (curl)

If you're comfortable with the command line, you can use the `curl` command to make calls to the API as well. This tool is fairly straightforward and makes it possible to interact with the API more completely, using all of the available methods rather than

limiting transactions to read operations as the browser does. `curl` is native on Unix-based systems such as Linux and Macintosh, and you can install it easily for Windows from http://curl.haxx.se/download.html.

Let's take a quick tour through the API using `curl`. By default, `curl` uses `GET` (read), but you can specify other methods on the command line, as shown in the following examples. Remember that your responses may be different if other people have been changing things; just go ahead and work with what you get. Don't be shy—this API is just for this book, and you can't break anything important. The best way to understand this type of system is to work with it yourself.

First, let's use `curl` to look at a single topping. Lines beginning with a dollar sign indicate a command-line call. The other information is the information returned by the server itself.

Listing 2.3 GET /api/v1.0/toppings/1

```
$ curl http://irresistibleapis.com/api/v1.0/toppings/1
{
  "topping": {
    "id": 1,
    "title": "Pepperoni"
  }
}
```

That seems pretty reasonable. I'd eat a pizza with pepperoni on it. Let's list all the toppings and see what else is on the pizza. Remember that the list for the toppings is at the parent level, or `/api/v1.0/toppings`.

Listing 2.4 GET /api/v1.0/toppings

```
$ curl http://irresistibleapis.com/api/v1.0/toppings
{
  "toppings": [
    {
      "id": 1,
      "title": "Pepperoni"
    },
    {
      "id": 2,
      "title": "Pineapple"
    },
    {
      "id": 3,
      "title": "Pickles"
    }
  ]
}
```

Wait, what? Pickles? That's kind of gross. Let's delete that one. The ID for it is 3, so the correct path to operate on is `/api/v1.0/toppings/3`.

Listing 2.5 DELETE /api/v1.0/toppings/3

```
curl -i -X DELETE http://irresistibleapis.com/api/v1.0/toppings/3
{
  "result": true
}
```

The response here says we succeeded. Just to be sure, let's pull a list of toppings again.

Listing 2.6 GET /api/v1.0/toppings

```
$ curl http://irresistibleapis.com/api/v1.0/toppings

{
  "toppings": [
    {
      "id": 1,
      "title": "Pepperoni"
    },
    {
      "id": 2,
      "title": "Pineapple"
    }
  ]
}
```

Okay, that's much better. But our pizza has pepperoni and pineapple, and I'd much prefer ham with my pineapple. Let's go ahead and change that first one to make the pizza how I want it. To update an existing item, the command needs to send a `PUT` to the resource with the new information required.

Listing 2.7 PUT /api/v1.0/toppings/1

```
$ curl -i -H "Content-Type: application/json" -X PUT -d '{"title":"Ham"}'
http://irresistibleapis.com/api/v1.0/toppings/1

{
  "topping": {
    "id": 1,
    "title": "Ham"
  }
}
```

Nice, now the pizza is looking pretty good. But really, as far as I'm concerned the pizza is just a vehicle to get cheese in my mouth, so I'll add some extra cheese to go with the Hawaiian pizza I've built.

Listing 2.8 POST /api/v1.0/toppings/1

```
$ curl -H "Content-Type: application/json" -X POST -d '{"title":"Extra extra
 cheese"}' http://irresistibleapis.com/api/v1.0/toppings
{
  "topping": {
    "id": 3,
```

```
    "title": "Extra extra cheese"
  }
}
```

Let's do one final check to make sure that the pizza looks good.

Listing 2.9 GET /api/v1.0/toppings

```
$ curl http://irresistibleapis.com/api/v1.0/toppings

{
  "toppings": [
    {
      "id": 1,
      "title": "Ham"
    },
    {
      "id": 2,
      "title": "Pineapple"
    },
    {
      "id": 3,
      "title": "Extra extra cheese"
    }
  ]
}
```

Awesome! Now the pizza is just right.

Note that with `curl` you can also pass `-i` for slightly more chatty information, or `-v` for a much larger dose of verbose output. If you're having fun and you'd like to try those now, feel free. The extra details you'll see are HTTP transaction details, which will be described in chapter 4.

2.4.3 HTTP sniffers

Browsers have become very capable at showing information about the calls they're making, but this is of limited use for a couple of reasons. As I mentioned earlier, a browser is really only capable of sending a read request, which restricts the actions you're able to explore. When you submit a form, it will create a create (`POST`) request, but you don't have the ability to arbitrarily call these operations in your browser.

HTTP sniffers are tools that allow you to explore all the HTTP traffic your system processes. HTTP sniffers watch and report on the network traffic your system is generating, whether it comes from a browser, an application, or a raw command-line call. With these tools, you can see the entirety of the HTTP request and response, and this allows you to debug what's happening if you're running into issues.

If you're using a Mac, HTTPScoop (www.tuffcode.com) is a very friendly choice. It's easy to set up and use, and the output is clear and complete. The downside to this tool is that it can't monitor secure transactions (HTTPS calls) and so it's not going to work with any API requiring secure calls. For the purposes of this book, though, you'll only be accessing a nonsecure API (the demo API) so HTTPScoop is a fine choice—it

would be my first choice for any Mac users wanting a reasonably intuitive experience. The license cost is $15 but you can try it for two weeks for free.

Figure 2.6 shows an example of the windows in HTTPScoop. For this chapter, I'll focus on the main screen listing all calls and the Request/Response tab. Later in the book you'll learn about headers, status codes, and other HTTP details so you can understand how they all interact together. For now, however, consider the request to be a simple request and response, and don't worry about the particular details if you're not already familiar with HTTP.

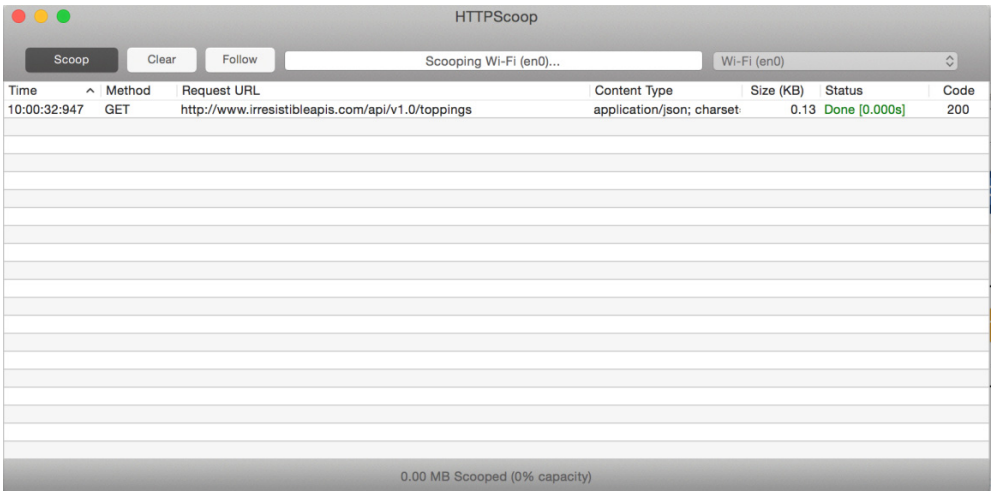


Figure 2.6 This is an example of a call being inspected by HTTPScoop. On this basic landing page, you can see the Request URL, representing the resource. The content type of the response, status code, and response size are also provided.

For Windows users, the best choice out there is Fiddler, which you can find at www.telerik.com/fiddler. For Windows, Mac, and Linux, there's a slightly more complicated choice in Charles (http://www.charlesproxy.com). If you're quite advanced in your network administration skills, you can try out Wireshark from https://www.wireshark.org. Wireshark is available and free for every major platform and sniffs all kinds of traffic, not just web/HTTP traffic, but the interface is quite complex and it can be difficult to understand what you're seeing.

EXERCISE 1 Watch the traffic in an HTTP sniffer as you go through the exercises from this chapter. Use the `curl` calls to access the API directly and see what the calls look like. For more verbosity with `curl`, you can use `-v` in your command and see more information about the call from the client side. Compare the information in the sniffer to what `curl` sends and see if you can find patterns. Which debugging method gives the best information? Which one is easier for you to use?

EXERCISE 2 Make a deliberately incorrect call. Call `/api/v1.0/toppings/100`—there's not likely 100 toppings on the pizza so this is a bad call. What kind of output did you get from `curl -v`? What did the HTTP sniffer show? The status code tells you how the system responded, which should give you the information you need to figure out what the issue is.

2.5 Interaction between the API and client

Seeing these `GET` calls to the API is somewhat interesting, but unfortunately you can't see the `POST`, `PUT`, or `DELETE` calls using a browser. `curl` isn't very intuitive for exploring a system. Without some kind of application using the API, it's difficult to explore and visualize the elegance and simplicity of this kind of interface.

Keeping in line with the simple API, I've created a simple application to exercise the API, creating a list of toppings for your virtual pizza. Again, for a real application there would be a full pizza and a method to place the order, but this application is deliberately as simple as possible so it's easy to understand how it works.

I'll go through the same sequence I did in the last section. Here's our starting pizza, with pepperoni, pineapple, and pickles. Loading the initial page causes an API call to be generated and we get the current list of toppings from the system.

First, take a look at the JSON representation returned when the API is called directly at `/api/v1.0/toppings`, shown in figure 2.7. Figure 2.8 shows how the application looks when this API call is made on the back end.

```
{
  "toppings": [
    {
      "title": "pepperoni",
      "id": 1
    },
    {
      "title": "peppers",
      "id": 2
    },
    {
      "title": "pickles",
      "id": 3
    }
  ]
}
```

Figure 2.7 Here you see a representation of the API toppings list in JSON, the markup language used by the platform. As described, the curly braces indicate an object, or dictionary, and the square brackets represent an array, or list of objects.

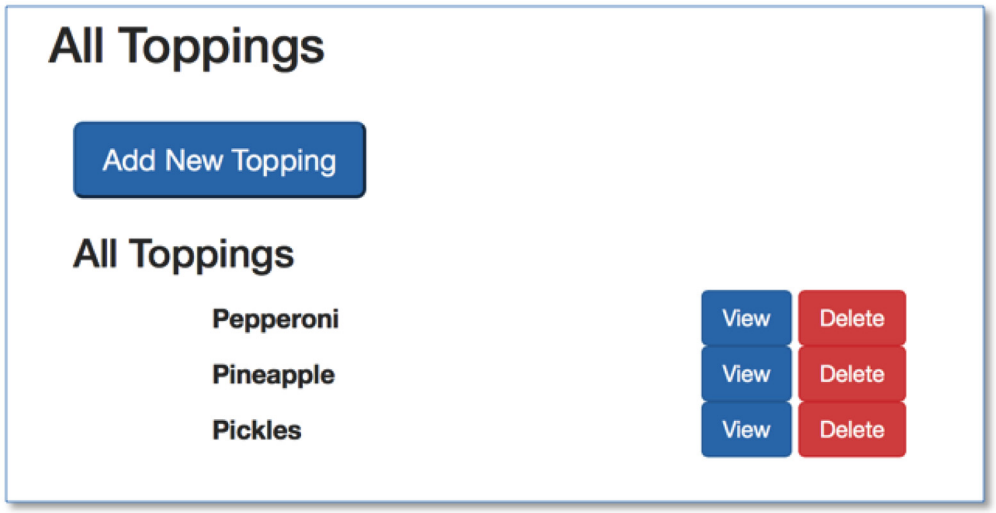


Figure 2.8 The application view for the toppings list shows the same information as shown in figure 2.4. This screen is created by calling the toppings list and creating the HTML based on the returned information. So if the list changes on the server, both figure 2.4 and figure 2.5 would change, with both showing the same information in different ways.

Now take a look at the main application at http://irresistibleapis.com/demo. With the JSON data, the simple application can build the front page. Some of the items are static—that is, they don't change. The top half of the page, for instance, is always the same, with the title of the display and a button to add new toppings. The bottom half, though, is created based on the information retrieved from the API. Each topping is listed, and the ID of the topping is used to create an appropriate button to act on that specific item. The user has no need to understand the relationship between the ID and the name of the topping, but the IDs used programmatically to set up the page to be functionally correct. Note how the information in the API in figure 2.4 directly maps to what's shown in the application in figure 2.5. The buttons on this page map directly to the other API calls, as shown in table 2.2.

Table 2.2 The mapping between the API calls and application functions

API call	Application function
`GET /api/v1.0/toppings`	Main application page
`GET /api/v1.0/toppings/1`	View button on main page
`POST /api/v1.0/toppings`	"Add new topping"
`DELETE /api/v1.0/toppings/1`	Delete button on either page

As we walk through the API actions, use the HTTP sniffer of your choice to watch the traffic as the interactions happen. A note here: Because this system is live, other people may have added, deleted, or edited the toppings and they may not match. Feel free to use the buttons to adjust the toppings to match, or just follow along with your own favorite toppings (Jalapeños? Sun Dried Tomatoes? Legos?).

The first action in the previous example was removing the pickles from the pizza, and clicking Delete on this page for the Pickles entry will do just that. This button knows which ID to operate on because it was embedded in the page when the listing was rendered.

Clicking the Delete button will make the `DELETE` call and then make a call to the API to re-render the list of toppings with the deleted topping gone. If you're using an HTTP sniffer or have configured your browser to show you web traffic, you can see this call happening from your system. Figure 2.9 shows what it looks like in HTTPScoop.

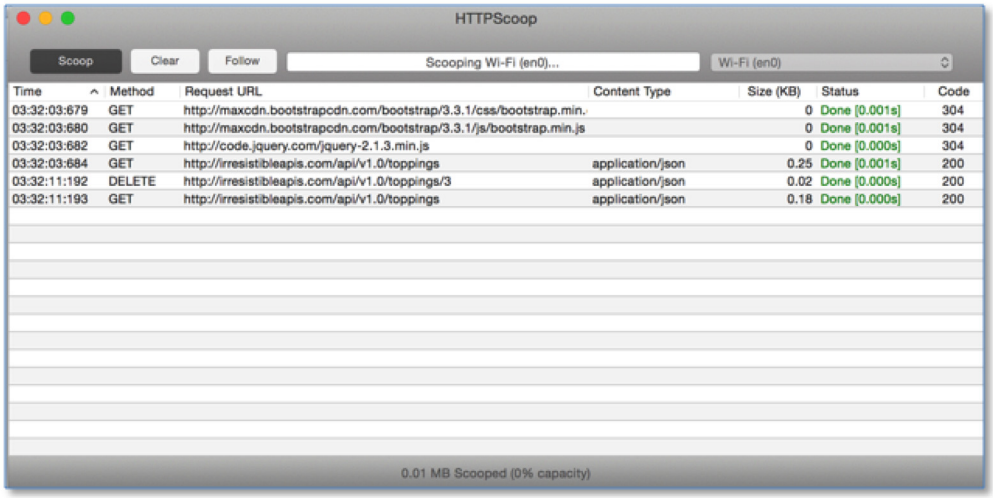


Figure 2.9 This HTTPScoop screen shows a list of all the calls made by the system. In this case, you can see the `DELETE` method is called to remove the `/toppings/3` resource from the system, and it was successful, as indicated by the 2XX response in the code column.

As you can see, the application pulled a few different framework files, and then got the full listing for the main page. When I clicked Delete, the application sent a `DELETE` request to the API server and then requested a new list of toppings. All of the requests were successful, so the main page refreshed to show the new list. Figure 2.10 shows the list after I deleted the offending pickles from the toppings list.

To edit an existing topping, in this case to change Pepperoni to Ham, click the View button. Doing so makes the read call for the specific item and allows you to edit the title. Using this technique to edit the Pepperoni to Ham, and then clicking Save, causes a `PUT` to happen exactly as in the original example. Watch your HTTP sniffer or browser traffic to see how this progression works. Figure 2.11 shows what the edit page

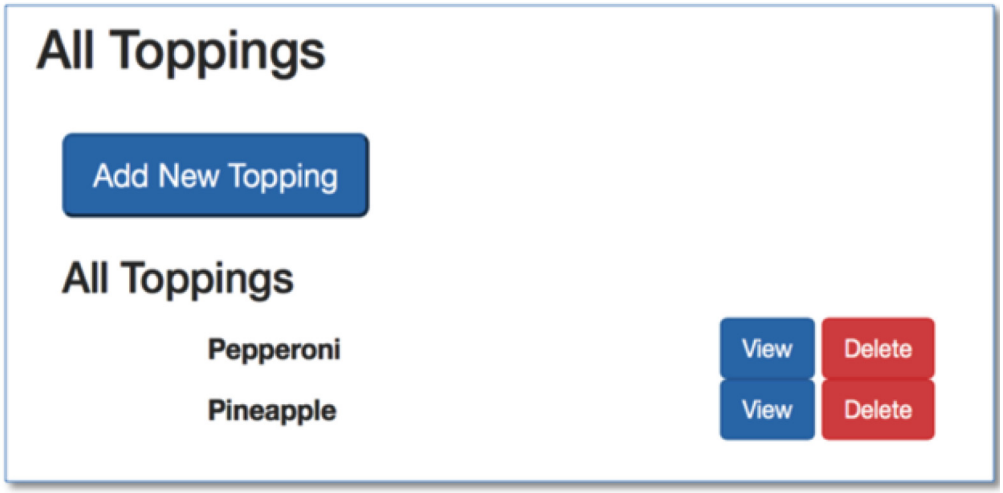


Figure 2.10 Once the topping has been deleted from the system, the HTML representation of the toppings list no longer shows the deleted topping. If the platform call is made (to `/toppings`) you'll see that the change is reflected in the JSON representation as well.

looks like for a particular topping—in this case I changed the title from Pepperoni to Ham. When this change is `PUT` to the API, it will change the item's title from Pepperoni to Ham, updating the database to reflect the change permanently.

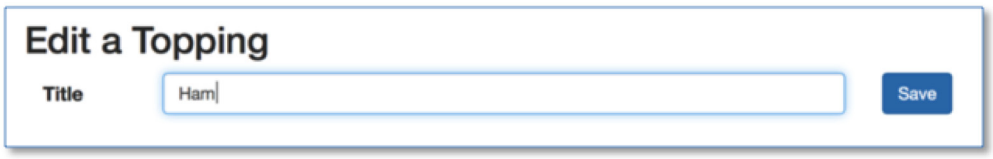


Figure 2.11 The Edit a Topping screen allows you to change the title of an existing resource.

The `PUT` request, viewed in HTTPScoop, shows the request and response (see figure 2.12).

As with the associated `curl` request earlier, the debugging demonstrates that the client sends a request including the new information for the requested item. A `PUT` request replaces information for an existing item in the system. In the response, the server returns a response showing the new values for the resource. This returned object matches the object that was `PUT` to the system. Without HTTPScoop this seems a little magical, but you should be seeing a pattern by this point; these common operations are direct mappings to system calls on the back end of the application.

Again, once the topping is edited, the application redisplays the main page, now with Ham and Pineapple (figure 2.13).

What's left then? Now I need to add my extra cheese to the pizza, because it's my favorite sort of thing. Clicking the Add New Topping button on the main page gives

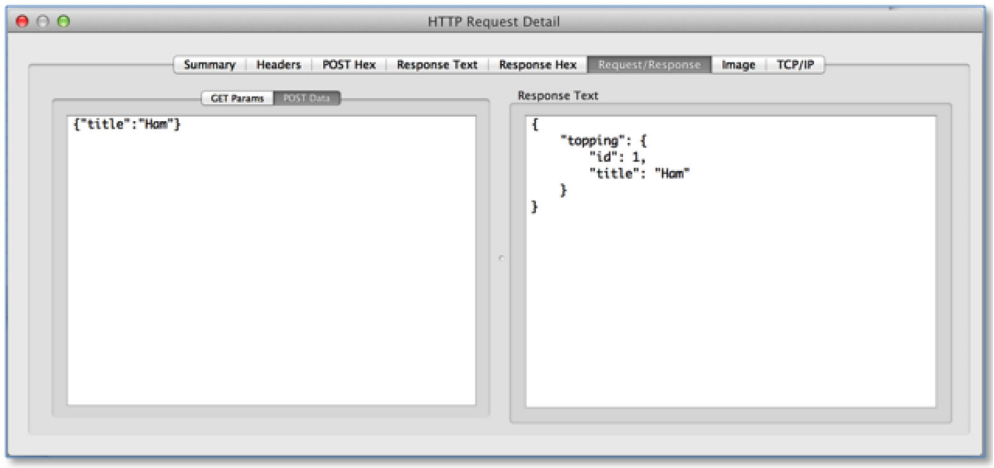


Figure 2.12 When you change the title of an existing resource, the information is sent to the server, and it sends back the new representation of that item. In this case, the object is quite simple; the title is the only field that can be changed. This is a simple demonstration of how an update works on an API platform.

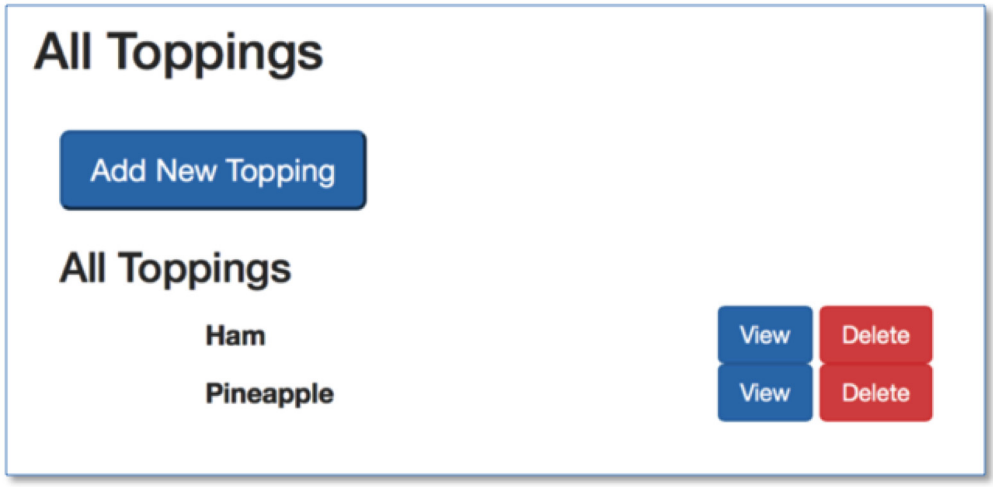


Figure 2.13 The list of toppings now includes Ham and Pineapple; the Pickles have been deleted (thank heavens) and the Pepperoni has been changed to Ham using an update. Again, if you made a call to the `/toppings` resource you'd see the changes shown in the JSON representation as well.

me a page for adding a new topping, as shown in figure 2.14. Remember, adding a new item to the list is a `POST` action, and that's what will happen on the back end. Figure 2.15 shows what the API transaction looks like when this `POST` is sent.

This example demonstrates again the difference between `PUT`, which updates a specific existing item, and `POST`, which creates a new item by adding it to the specified list.

Figure 2.14 The Add a Topping screen is designed to add new toppings to the system. As mentioned earlier, a create action is generally represented by a `POST` operation, and that's what the system will do in this case.

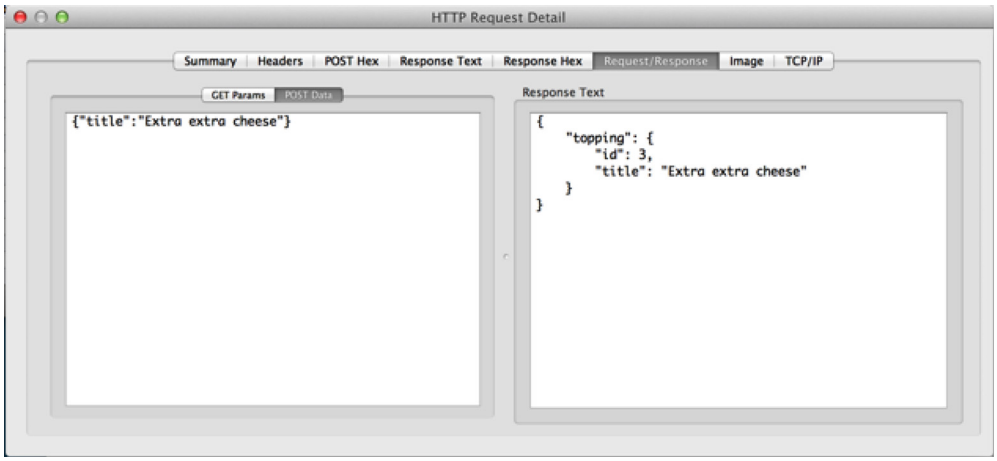


Figure 2.15 HTTPScoop `POST` request/response. The only field needed to create a new topping is the title, and it's set to `Extra extra cheese` (yum!). The response shows the ID and title—the entire representation—of the newly added item.

After adding this new topping to the system, the application again requests the list of toppings, which brings the web page back, once again, to the main page. This completes the circuit using an application to exercise the back-end API. The single page running this application is quite straightforward, because all the logic and actions are happening on the back end using the API.

Now that you've had the opportunity to view some specific traffic, take time to play with the example application with the various HTTP inspection methods. Because this sample application runs in your browser, you have the option of using developer tools in your browser to watch the traffic or an HTTP sniffer for this exploration. For the exercises in this book, you'll want to use an HTTP sniffer, so pick the one you're most comfortable with and start familiarizing yourself with its use.

Advanced Example Note

If you're a developer and wish to install your own copy of this system, follow the instructions in section 2.6 to do so. Otherwise, skip to section 2.7 for a summary of this chapter.

2.6 Install your own API and front end

This optional section is designed specifically for developers who want to understand more completely the back-end functionality of the API and sample application. You can use a Docker container to run the system quickly on your own system or download the code from my GitHub repository. I'll walk through the steps to install and use the Docker container first, and then give more general instructions for grabbing the code from GitHub to run on your own system.

2.6.1 Installing the system via Docker

Docker is extremely simple to install on Linux systems, and quite easy on OS X and Windows systems as well. Installing the Docker container is simple once you've got the Docker system set up. Using this container allows you to separate the code and processes from the main processes on your system while avoiding the memory and space management issues of more heavyweight virtual machine systems.

The Docker installers for installation on Windows and Macintosh are here:

```
https://www.docker.com/toolbox
```

If you're an advanced user running Windows and already have virtualization working via VBox or another virtualization system, you need to be aware that Docker relies on VirtualBox, which may conflict with your existing setup. Additionally, boot2docker requires that virtualization be available on your system, which infrequently requires changes to the BIOS. Also, virtualization is only available on 64-bit systems. If your system is a 32-bit system, you'll need to install the code directly from GitHub.

Once you've installed Docker using the instructions at the Docker website, you're ready to pull and run the container.

On Linux, issue the command (on one line)

```
% sudo docker run -p 3000:3000 synedra/irresistible
```

to bind your system's port 80 to the Docker container on port 3000.

On systems using boot2docker (Windows or Mac OS X), the command is as follows (root access isn't needed because of the nature of docker-machine):

```
% docker run -p 3000:3000 synedra/irresistible
```

The application automatically runs in the Docker container. When using boot2docker, the Docker engine assigns a separate IP address for Docker containers. In order to determine the IP address of your Docker container, issue the command `docker-machine ip default`. Once you've done that, you can access the system at http://<docker-ip/. Because the server is running on port 80, the default web port, the browser will find the web server on that port.

If you'd like to start the container and explore the code, you can do so with the following command, which won't start the node server:

```
% docker run -i -t synedra/irresistible /bin/bash
```

You'll now be root in a shell within the container. Accessing the system in this way allows you to look at the code and figure out how all the pieces are working together. The application itself is composed of the toppings.js file, and the front-end web server is run from the static/index.html file. The previous command will allow you to access the application directly without cross-domain issues. You can read more about Docker port forwarding at https://docs.docker.com/userguide/dockerlinks/.

If you're running Docker directly on Linux, you can access the system directly at http://localhost. If you already have a web service running on the default port, you can assign a different port in the `docker run` command.

2.6.2 Installing the system via Git

If you prefer to run the applications on your own system rather than using the Docker container, you need to have Git and Node.js installed on your system. The commands needed to pull the repository to your system and install and run node are as follows:

```
% git clone https://github.com/synedra/irresistible
% cd irresistible/
% curl -sL https://deb.nodesource.com/setup | bash - && apt-get
 install -yq nodejs build-essential
% npm install -g npm
% npm config set registry http://registry.npmjs.org/
% npm install -g express@2.5.1
% npm install express
% npm install
% node toppings.js
```

From there you can access the system at http://localhost:3000 (or port 3000 on whichever server you're using). Node.js runs on port 3000 by default, so if you want to expose the system on the standard port (80), you'll want to run a separate server on the front end—something like Nginx or Apache—and then create a reverse proxy back to the node server. For security reasons it's best not to use root to run a bare web service, and you can't access the standard ports as a regular user. This is one of the advantages to using the Docker system—because it's isolated from the rest of your system at its own IP address, it's safe to run the front-end server on port 80.

2.6.3 Exploring the code

As you're running the system and exploring it, you'll see the logs for the system show up in the terminal window where you started up the web server. Using an HTTP sniffer, you can watch the API traffic your system is generating as described in section 2.3. Once you've started a web browser at http://docker_ip_address/, not only will you be able to see the traffic in an HTTP sniffer, but you'll start seeing server entries in the terminal window that you started.

The logs show you all the traffic—both front-end calls to / and the back-end requests to the API. This combined log data makes it easy to see how the systems are interacting.

If you used the Docker setup, you were placed directly into the /opt/webapp directory. The Git instructions will put you in the same directory: the webapp subdi-

rectory of the repository. Table 2.3 shows a listing of the files in the program directory along with a description of what each one does.

Table 2.3 Files included in the program directory

Filename	Description
Procfile	Used if you want to deploy this to Heroku
Toppings.js	The main program for the system
static/index.html	A very simple single-page application that exercises the API

The toppings.js file is used to run the node web server. When you type `node toppings.js` the application looks for the index.html file in the static directory and serves it up.

The application uses Bootstrap, a single-page application framework that makes your simple applications look pretty. The formatting pieces are mostly contained within the Bootstrap framework, and overrides are made within the index.html file. This is all to explain what the `id` and `style` attributes are for each `<div>` on the page. In this case, it's using the main-single-template for the outside wrapper, and the inside is a main-single container. This function will present the table of items for the page to render.

The `$.get` function makes the call to `/api/v1.0/toppings`, at which point the back end returns a list of toppings, and this function is called to render the page.

EXERCISE 3 Play around with the page, see how each piece works, and try to see if you can make the application go directly to the Edit page from the toppings list instead of the View page.

2.7 Summary

At this point you've either played directly with my hosted service or set up your own. This chapter covered the following concepts:

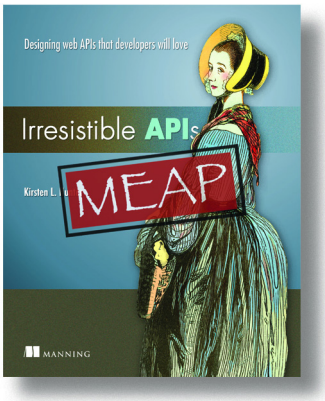
The structure of a simple web API system includes the required actions for a complete platform: create, read, update, and delete.

A basic HTTP transaction includes a clearly defined request and response, creating a foundation for web APIs.

From HTTP sniffers to Chrome Developer Tools, the ability to monitor the traffic makes it much easier to understand what's happening between the systems.

RESTful API ideals define the endpoints as nouns, and not verbs. Between these ideas and the HTTP transactions they work with, the web API system is complete.

Now that you have an understanding of the various moving pieces in a simple API, you can begin thinking about your own API at a higher level: how to architect the entire system to use the simple pieces I discussed here to build a fantastic API system. This chapter was more about the bottom up, and how the cogs and wheels work together to make things work. The next chapter will help you to learn how to think top down: what are the goals for your API system and how can you meet them most efficiently?



Irresistible APIs provides step-by-step guidance for designing APIs that reflect an application's core business value, delight the developers who use them, and will stand the test of time. In it, business product managers and developers learn to treat an API as a first class product. You'll discover what questions to ask during design so that the first version is the best possible product. Because APIs are a combination of a business need and a technical implementation, exercises throughout the book present both sides of the design process, so that you can engage with the material that's most comfortable and relevant for you. When you finish, your team will be able to design APIs that attract developers, lead your industry, and add value to your core business.

A Web API is a platform with a web-style interface developers can use to implement functionality. Well-designed APIs feel like a natural extension of the application, rather than just a new interface into the backend database. Designing Web APIs based on use cases allows an organization to develop irresistible APIs, which developers can consume easily and which support the business values of the organization. By providing clear, jargon-free guidance on Web API design principles and processes, this book breaks down the artificial barriers between business product managers and developers implementing APIs.

What's inside

- Step-by-step guidance through the API design process
- Design Driven Development
- Tips for setting up your API design team
- The role of API design tools

Written for all members of an API design team regardless of technical level.

Communicating with the Server

The Single Page Application (SPA), an increasingly common form of client software for web applications. This design typically includes much richer browser-based UI code and relies heavily on frameworks and web APIs. This chapter considers how SPAs typically interact with RESTful web APIs.

Chapter 7 from *SPA Design and Architecture: Understanding single-page web applications* by Emmit A. Scott, Jr.

Communicating with the server

This chapter covers

- The server's role in an SPA environment
- How MV* frameworks communicate with the server
- Handling results with callback functions and promises
- Consuming RESTful services

In chapter 1, you learned how the adoption of the XMLHttpRequest (XHR) API and the AJAX movement eventually led to the emergence of SPAs. After XHR was supported in the browser—as a COM component at first and then natively—developers could use it to asynchronously load both the application's scaffolding and its data without refreshing the page. This opened many new avenues for the ways that web pages could be constructed.

Until now, you've been focusing on creating the SPA itself. In doing so, you've used XHR to dynamically retrieve the templates used to construct your views but restricted the data in your sample applications to local stub data. In this chapter, you'll take another important step forward. You're going to move the source data to the server and learn how to remotely access it from your SPA.

We'll kick things off with a brief look at the communication process between the SPA client and the server. After you're clear on the overall process, we'll look at the details of what happens on the client.

On the client side, I'll focus on how MV* frameworks try to make your life easier when you need to talk to the server. MV* frameworks that have built-in support for persistence enhance the XMLHttpRequest API with their own expanded set of features. But because each one has to go through XHR, there are certain commonalities I can point out.

What's the optimal way for an SPA to communicate with the server?

Generally, the most optimal way to communicate with the server from your SPA is to use the objects provided by your MV* framework—provided that the framework supports server communication. Because its objects are built specifically to work within the framework, they provide request and response methods that are ready-made to work with the rest of the framework. You'll need to customize these objects for your specific needs, either through configuration or by extending them in some fashion. Typically, you don't need to supplement the framework with any additional libraries.

If your framework doesn't have built-in support for communicating with the server, you can opt to work directly with the low-level methods of the `XMLHttpRequest` object itself, use a general utility library (such as jQuery), or go for a library that has fewer, more specialized components (such as AmplifyJS, http://amplifyjs.com).

After learning the basics of communicating with the server, you'll turn your attention to dealing with the results. You'll start with traditional callback functions, which describe what you want to happen when calls succeed or fail. Next, you'll learn about the use of promises. Promises are fast becoming the preferred means of dealing with XHR results by many of today's MV* frameworks. More important, though, they're part of the ECMAScript 6 version of JavaScript. They're generally considered a cleaner, more elegant way of dealing with asynchronous processes than simple callback functions.

This chapter wraps up with a look at consuming RESTful services with your SPA. REST is an architectural style for both websites and web services that has gained widespread popularity in recent years—so much so that many MV* frameworks support it out of the box. Some even use the REST style as a default.

I won't go into great detail about designing RESTful services, because that server-side topic is beyond the scope of this book. I'll talk about what REST is in the philosophical sense and discuss some of the ways in which MV* frameworks approach REST.

In the example for this chapter, you'll continue the preceding chapter's used video game store project by adding a shopping cart to it. A shopping cart is a standard feature for most sites selling goods and/or services online. It's also the perfect venue for demonstrating the server communication concepts in this chapter. You'll explore the details of your shopping cart later in the book. That being said, the sample project has some new requirements that we need to discuss.

7.1 Understanding the project requirements

Unlike in previous chapters, to run the code in this chapter you'll need a server. Because most MV* frameworks, including the one we're using (AngularJS), are server agnostic, you can pick any server you want. You can also use any server-side language you want. So whether you like JavaScript, PHP, Python, Ruby, .NET, Java, or any other of the multitude of languages out there for server-side development, that's perfectly OK.

Here are the only two hard requirements for whichever server/language combination you prefer:

- Support of RESTful services, because the example uses REST
- JSON support, either built in or via an add-on

The example's server code was developed using Spring MVC (version 4), which is a Java-based MVC framework. Don't worry, though, if you don't know Java or Spring. In our discussions within the chapter, I'll refer to the server-side code only conceptually. A guide to the server-side code's configuration is available in appendix C. If you prefer a different server-side tech stack, the appendix begins with a summary of the server-side objects and tasks so you can structure your own server-side code accordingly. The entire source for the project is available online for download.

Now that you've been introduced to the project, let's look at how your SPA can communicate with the server.

7.2 Exploring the communication process

Though many concepts around communicating with the server are the same for any type of web application, the next few sections present some of the basics within the context of a single-page application. I'll also highlight some specific ways in which the MV* framework supports the communication process.

7.2.1 Choosing a data type

In order for the SPA running in the browser to communicate with a server, both need to speak the same language. The first order of business is deciding on the type of data that will be sent and received. To illustrate, I'll use the example of a shopping cart, as I do at the end of the chapter.

When the user interacts with your shopping cart—whether it's adding an item, updating the quantity of an item, or viewing the current state of the cart's contents—you're sending and receiving JSON-formatted text. JSON is commonly used by SPAs when communicating with servers, though the data type can be anything from plain text, to XML, to a file.

Even though you're using JSON-formatted text as a common data exchange format, it's merely a representation of a system's native object or objects. For the text to be useful, conversions are happening at both ends. You'll learn about these a little later. To ensure that the conversions to native objects work, each side must do its part to make sure the agreed-upon JSON format is used in the call.

When a call is made to the server, requests can include information about the *internet media types* that are acceptable, because a resource can be available in a variety of languages and media types. The server can then respond with a version of the requested resource that it deems a best fit. This is called *content negotiation*. For this project, you're interested only in JSON. To express this, you can explicitly declare an internet media type of `application/json` for the exchange.

> **Internet media types**
>
> An *internet media type* (formerly a MIME type) is a standard way to identify the data that's being exchanged between two systems. It's used by many internet protocols, including HTTP. Internet media types have the format of `type/subtype`. In this case, you're using a media type of `application/json`: the type is `application`, and the subtype is `json`.
>
> Optional parameters can also be added by using a semicolon, if required. For example, to specify a media type of `text`, with a subtype of `html` and a character encoding of `UTF-8`, you use `text/html; charset=UTF-8`.

Internet media types are specified using *HTTP headers*, which are the fields sent in the transmission that provide information about the request, the response, or what's contained in the message's body. The `Content-type` header tells the other system what to expect in the request and response. The `Accept` header can also be specified in the request to let the server know the media type or types that are acceptable to return.

After a data type has been selected, an appropriate request method must be used for the call to be successful. The next section presents common request methods for an SPA.

7.2.2 Using a supported HTTP request method

When a client makes a request, it can indicate the type of action it would like the server to perform by specifying the *request method*. In order for the request to be successful, though, the HTTP request method specified in the request must be supported by the server-side code for that call. If it isn't, the server may respond with a 405 Method Not Allowed status code.

Because the HTTP request method describes what should happen to the resource represented in the request, it's often called the *verb* of the call. A request method that doesn't modify a resource, such as `GET`, is considered *safe*. Any request method that ends in the same result, no matter how many times its call is executed, is considered *idempotent*. For example, you'll use `PUT` when the user wants to update the count of a particular item that's in the cart. Because `PUT` is idempotent, you can tell the server that you want two copies of *Madden NFL* 10 times in a row, but after the tenth time, you still have only two copies in the cart.

Table 7.1 defines a few common HTTP request methods used in our shopping cart example. Although it's not a comprehensive list, it does represent the ones most commonly used in single-page applications.

Table 7.1 Common HTTP methods used in an SPA

Method	Description	Example	Safe?	Idempotent?
`GET`	Typically, `GET` is used to fetch data.	View the shopping cart	Yes	Yes
`POST`	This method is most commonly used for creating a resource or adding an item to a resource.	Add an item to the cart	No	No
`PUT`	Typically, `PUT` is used like an update-or-create action, updating the existing resource or optionally adding it if it doesn't exist.	Update the quantity of an item in the cart	No	Yes
`DELETE`	This is used to remove a resource.	Remove an item from the cart	No	Yes

Other HTTP methods are specified in the HTTP protocol. For a full list, see http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods.

The final part of the communication process is the conversion of the data to and from the internet media type sent and received.

7.2.3 Converting the data

After the data type is agreed upon, both the client and the server must be configured to send and receive that particular type. For your shopping cart, you're using JSON exclusively, so both the code in the browser and the code on the server must be able to convert to and from this text format.

On the client, the ability to convert a JavaScript object to JSON may be built into the MV* framework. If that's the case, it's likely the default, and the conversion will happen automatically when you use the framework to make the server request. If automatic conversion is not built in, the framework may offer a utility for the conversion of its custom types. For the conversion of JavaScript POJOs, you can use the native JavaScript command `JSON.stringify()`:

```
var cartJSONText = JSON.stringify(cartJSObj);
```

On the server, the JSON-formatted text is converted into a native object of the server-side language by a JSON parser that's either built in or available via a third-party library. Like the HTTP method, the exact method for executing the conversion process on the server will vary.

To illustrate the process end to end, let's use the shopping cart update example again. Let's say that the user has increased the quantity of an item in the cart. For the modifi-

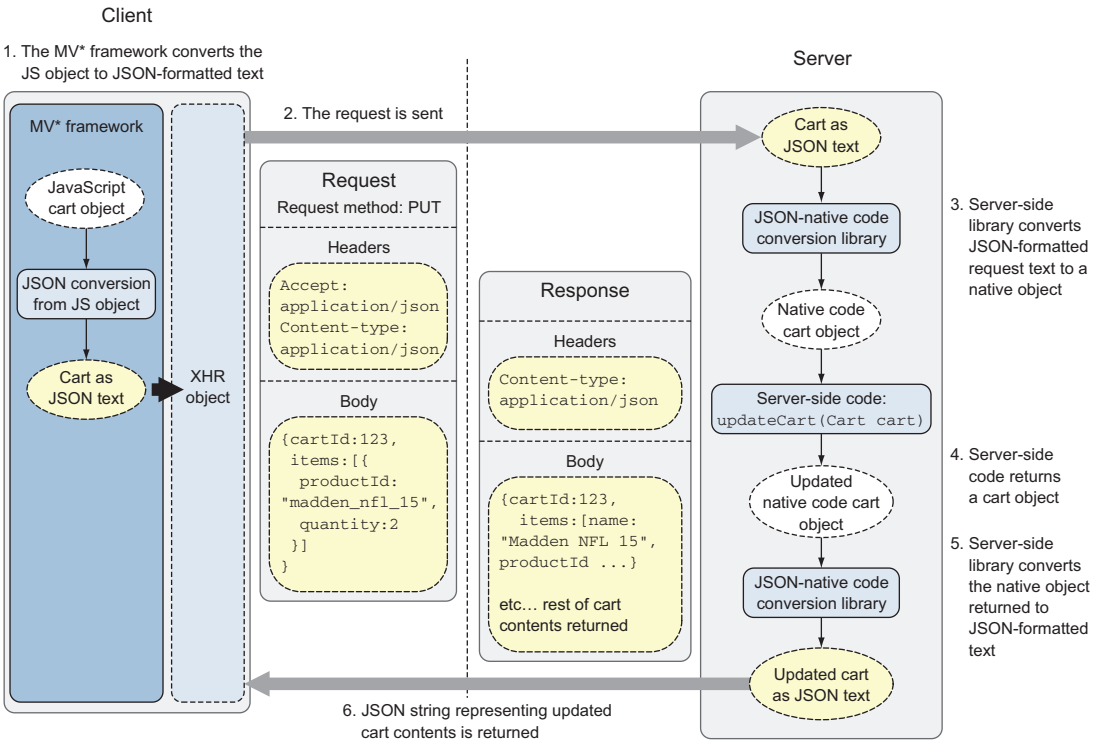


Figure 7.1 JavaScript objects are converted to JSON and added to the request body for the request. In response, the server sends back the updated cart as JSON via the response body.

cation to be verified and processed, you'll send the updated cart to the server. Figure 7.1 paints a picture of the conversions that happen at both ends.

After the update function is called, your JavaScript cart object is converted into JSON-formatted text by the MV* framework. Next, the MV* framework passes the data to the XMLHttpRequest API. Then the JSON payload is sent in the body of the request to the server.

On the client, after the response is received, the returned text is converted once again. This time it's converted back into a native JavaScript object. Often this is also handled automatically for you by the MV* framework. If not, you can use the native JavaScript command `JSON.parse()`:

```
var cartJSObj = JSON.parse(returnedCartJSONText);
```

Now that we've discussed the communication process as a whole, let's go back to the client to talk about how MV* frameworks help simplify this process.

7.3 Using MV* frameworks

One thing MV* frameworks are great at is simplifying complex tasks by abstracting away a lot of the boilerplate code involved. This is certainly true when it comes to communicating with the server. This section specifically covers making requests and dealing with the responses. In our discussion, I'll point out some of the ways in which MV* frameworks help with the heavy lifting.

7.3.1 Generating requests

If server communication is supported by the framework, it may expose the XHR object directly or abstract some or all of the XHR functionality with its own proprietary objects. These custom objects act as wrappers around the `XMLHttpRequest` object either directly or indirectly via another library such as jQuery. They add value by hiding many of the tedious, repetitive tasks in making calls and processing the results.

Before you look at any MV* examples, let's put things into perspective by using vanilla JavaScript and the `XMLHttpRequest` object to make a server call. If you need a refresher on XHR, refer to appendix B.

We'll use the shopping cart again as an example. As you did earlier, you'll update the quantity of an existing item in the cart. Because you're updating the quantity of an item, you'll use the `PUT` HTTP request method. As you learned in the preceding section, `PUT` is commonly used in an update situation. To keep things simple, you'll use an abbreviated version of the cart data used in the project:

```
var cartObj = {
   cartId : 123,
   items : [ {
        productId : "madden_nfl_15",
        quantity : 2
   } ]
};
```

The following listing illustrates the plain JavaScript version of an update to the shopping cart using the `XMLHttpRequest` object directly.

Listing 7.1 Shopping cart update using `PUT` and XHR directly

```
var cartJSON = JSON.stringify(cartObj);

var xhrObj = new XMLHttpRequest();

xhrObj
.open("PUT","/SPA/controllers/shopping/carts",true);

xhrObj
.setRequestHeader("Content-Type","application/json");

xhrObj
.setRequestHeader("Accept","application/json");

xhrObj.send(cartJSON);
```

Convert JS object to JSON text

Create new instance of XMLHttpRequest object

Define the call properties

Set the content type

Declare the data type you'll accept

Send the request

In this example, you're not even handling the results. You'll tackle that in the next section. Even so, you have to deal with several of the low-level details. You have to manually set the content type and any other headers you need (such as `Accept`). Additionally, you have to manually convert the JavaScript cart object to JSON-formatted text.

Generally, if an MV* framework has out-of-the-box support for server communication, you'll most likely be generating requests from one of two types of objects: a model or some type of utility/service object. If the framework requires you to create an explicitly defined data model, you'll most likely perform server operations by calling functions on the model itself. If the framework doesn't have an explicit data model (the framework considers any source of data an implied model), you'll probably work through the framework's utility/service. AngularJS, for example, provides a couple of services for server communication: `$http` and `$resource`. You're be using `$resource` in the project, and you'll see it in action a little later.

MAKING REQUESTS VIA A DATA MODEL

With some frameworks (Backbone.js, for example), you explicitly define a data model by extending a built-in model object from the framework. By extending the framework's model, you inherit many capabilities automatically. This includes the built-in capability to perform the full range of CRUD (create, read, update, and delete) operations on a remote resource (see figure 7.2).

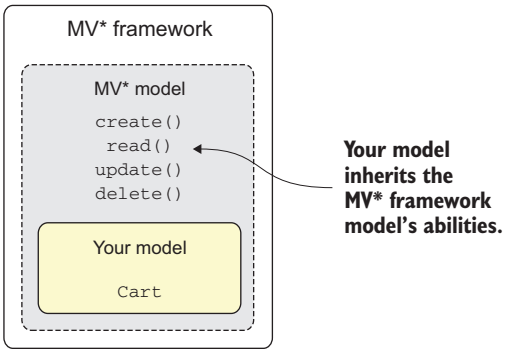


Figure 7.2 With MV* frameworks, where your model extends those of the framework, you automatically inherit abilities from the parent, such as the ability to make server requests.

Don't worry, though, if you need to make custom calls. Most frameworks let you override and customize their out-of-the box behavior.

Listing 7.2 extends `Backbone.Model` to define your shopping cart, passing in the name of the attribute you want to use as its ID. You're also defining a base URL for all server requests. You have to do this only once, because this is just the model's definition.

After the model has a definition, you can create new instances of it anytime you need to use it. All new instances of your shopping cart will then inherit everything you need for server communication.

Listing 7.2 Backbone.js version of your shopping cart update

```
var Cart = Backbone.Model.extend({
   idAttribute : 'cartId',
   urlRoot : 'controllers/shopping/carts/',
});

var cartInstance = new Cart(cartObj);
cartInstance.save();
```

Define a model with a URL and an ID

Create a new model instance

Call its inherited save() function to initiate the request

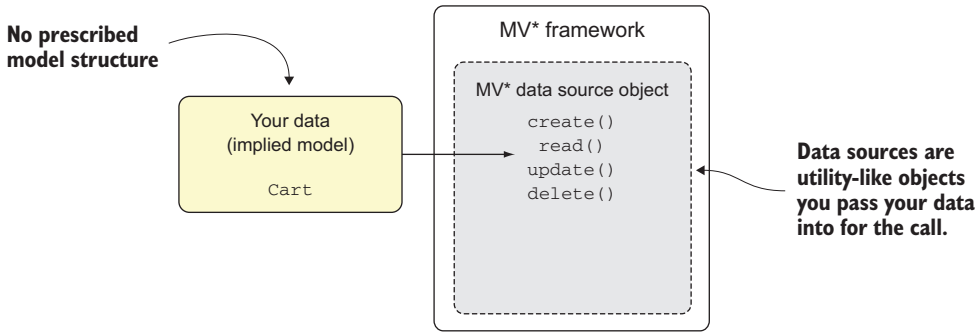


Figure 7.3 Frameworks that provide server communication, but don't provide a model to extend, will most likely provide a data source object instead.

The Backbone.js code is certainly less verbose. It's also doing several things under the covers. For starters, it assumes you're dealing with JSON (unless you tell it otherwise) and automatically converts the object passed into its constructor to JSON-formatted text. In addition, it automatically sets the `Content-type` and `Accept` headers for JSON. Finally, it can automatically decide whether to use `PUT` or `POST` based on whether the object of the request has an ID yet. Again, any of these features can be customized or overridden.

MAKING REQUESTS THROUGH DATA SOURCE OBJECTS

The other manner in which MV* frameworks make requests to the server is through a separate data source object. This is typical when a framework, such as AngularJS, allows you to use anything you want as a data model. With no parent to extend, there are no canned abilities to inherit. When this is the case, the framework provides a data source object that you'll pass your model into when making a call (see figure 7.3).

Let's see an example of this alternative MV* approach. Listing 7.3 uses an AngularJS `$resource` object to perform your shopping cart update. I mentioned earlier that `$resource` is one of AngularJS's services that can be used when communicating with the server. It has many features for easily modeling requests and dealing with the server's response. When you get to this chapter's project, you'll delve into the use of `$resource` in detail to understand the example. For now, let's see this style of MV* code as a comparison with your original, vanilla JavaScript server call.

Listing 7.3 AngularJS version of your shopping cart update

```
var CartDataSrc =
   $resource(
      "controllers/shopping/carts", null,
      {updateCart : {method : "PUT"}}
   }
);

CartDataSrc.updateCart(cartObj);
```

Define a URL for the call and no URL parameters (null)

Use the built-in `$resource` object to create a data source

AngularJS's $resource object has all CRUD operations except `update()`; you can configure

Use the updateCart() method you added

Even though you're not extending anything, the overall concept is the same as our first MV* example. You can lean on the MV* framework to help you generate the request. Like Backbone.js, under the covers AngularJS sets the appropriate headers, converts the JavaScript object into JSON, and uses the HTTP method you defined. As you saw, though, the authors of this particular framework chose to not include a method to update the cart (`PUT`) out of the box. It's easy enough, though, to customize the data source object to add this behavior.

Another feature that MV* frameworks provide is an easy way to deal with the results of a call to the server. Some frameworks support using callback functions, whereas others rely on promises. Promises are becoming more and more prevalent with MV* frameworks, but I'll make sure you understand using callbacks with asynchronous requests first.

7.3.2 Processing results with callbacks

When you're processing an asynchronous task, such as your server call to update the shopping cart, you don't always want the application to hang while you wait for the server to respond. You sometimes need it to continue in the background while your application handles other tasks. So instead of the update function returning a value when it's done, callbacks are passed in to handle the results when it completes. You can do this because functions can be passed around. This allows any function to take other functions as arguments.

When callbacks are passed into a function as arguments, they become like an extension of it. They can be passed control and continue processing from there. Using callbacks in this way is called *continuation-passing style*.

Let's take a look, then, at using the continuation-passing style of programming to process the results of a server call. Because Backbone.js supports callbacks, I'll use that framework to illustrate. Let's add some handlers to your previous shopping cart update. Figure 7.4 gives an overview.

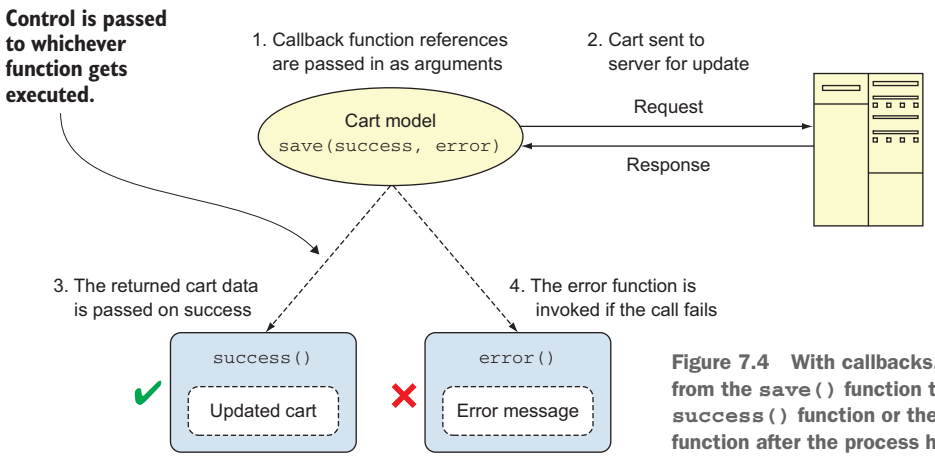


Figure 7.4 With callbacks, control passes from the `save()` function to either the `success()` function or the `error()` function after the process has completed.

If the call is successful, the `save()` function invokes the `success()` function via the XHR object, passing to it the returned cart data. If the call fails, `save()` invokes `error()`, passing in the details for the failure. In either case, processing is continuing from the model's `save()` function to one of these callback functions.

Now let's take a look at some code. You'll make exactly the same request that you did earlier with Backbone.js, but this time you'll do something with the results (see the following listing).

Listing 7.4 Processing a shopping cart update via callbacks

New Cart instance created

```
var cartInstance = new Cart(cartObj);

cartInstance.save(null, {
   success : function(updatedCart, reponse) {
      console.log("Cart ID: " + updatedCart.id);
   },
   error : function(cartUnchanged, response) {
      console.log("Error: " + response.statusText);
   }
});
```

No model attributes to change before saving (null)

Define callbacks for `save()`

Not only is this code a little easier to read, but you're also able to pass in a configuration object to the `save()` function itself. In this object, you can define success and error callback functions and any other configuration options supported by `save()` that are needed. Backbone.js also helps out by automatically passing the results of the server call to the callback functions you've defined.

In a successful call, you have access to the updated cart object as well as the response from the server. When the call fails, you can use the response to find out the reason for the failure. Moreover, if you need any low-level details about the call, the `save()` method also returns a jQuery jqXHR object, which is a wrapper for XHR. For more details about jqXHR, see http://api.jquery.com/jQuery.ajax/#jqXHR.

Callbacks are easy to work with and great for simple results, but continuation-passing style can sometimes become cumbersome if you have multiple tasks to perform when the call completes.

Fortunately, a trend with many MV* frameworks is to return a promise instead of relying on continuation-passing style callbacks. Like callback functions, promises are nonblocking: the application doesn't have to stop and wait for the call to finish. This makes them also ideal for asynchronous processing. As you'll see in the next section, they have additional properties and behaviors that make your life much easier when you have complex requirements for handling results.

7.3.3 Processing results with promises

A *promise* is an object that represents the outcome of a process that hasn't yet completed. When an MV* framework supports promises, its functions that perform asynchronous server calls will return a promise that serves as a proxy for the call's eventual

results. It's through this promise that you can orchestrate complex result-handling routines. To understand how to use a promise, you must first understand its internal state before and after the call is made.

WORKING WITH PROMISE STATES

The good news about working with promises is that they exist in only *one* of the following three states:

- *Fulfilled*—This is the state of the promise when the process resolves successfully. The value contained within the promise is the result of the process that ran. In your shopping cart update, this would be the updated cart contents returned by the server.
- *Rejected*—This is the promise's state when the process fails. The promise contains a reason for the failure (usually an `Error` object).
- *Pending*—This is the initial state of the promise before the process completes. In this state, the promise is neither fulfilled nor rejected.

These three states are mutually exclusive and final. After the promise has been fulfilled or rejected, it's considered settled and can't be converted into any other state. Figure 7.5 uses the shopping cart project to illustrate the three states of a promise.

A variable assigned to a promise doesn't remain a null reference while it waits for the function to return. Instead, a full-fledged object gets returned immediately in a pending state with an undetermined value. When the process finishes, the promise's

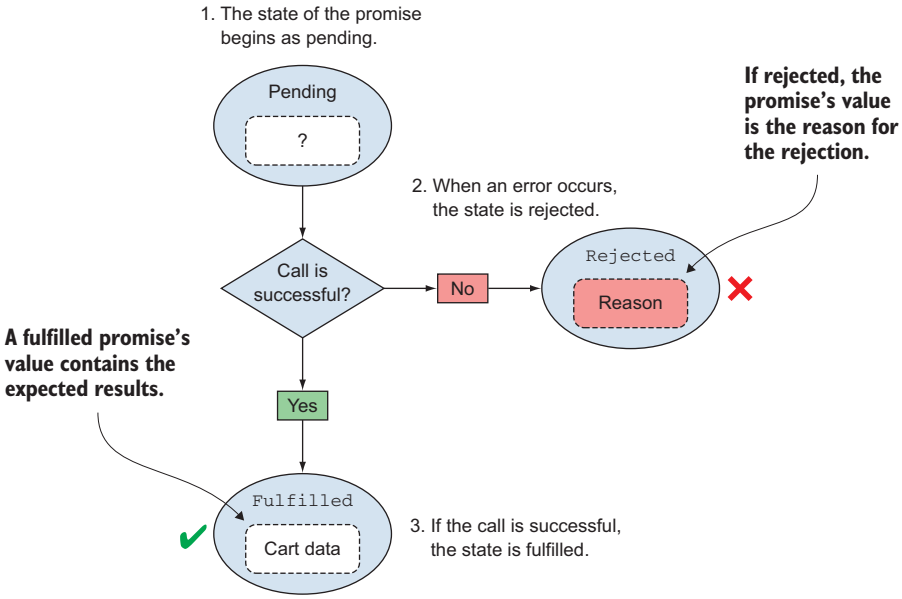


Figure 7.5 A promise has three mutually exclusive states: pending, fulfilled, and rejected.

state changes to either fulfilled, with its value containing the results of the call, or rejected, with the reason for the failure.

ACCESSING THE RESULTS OF YOUR PROCESS

I haven't talked about what you do with a promise after it's returned, in order to access a process's results. The Promise API has several useful methods, but the one you'll use the most is its `then()` method.

The `then()` method lets you register callback functions that allow the promise to hand you back a process's results. The functions you define here are called *reactions*. The first reaction function represents the case in which the promise is fulfilled. The second is optional and represents the case in which the promise is rejected:

```
promise.then(
   function (value) {
      // reaction to process the success value
   },
   function (reason) {
      // reaction to optionally deal with the rejection reason
   }
);
```

Because the rejected reaction is optional, the `then()` method can be written in shorthand:

```
promise.then(function (value) {
   // process the success value, ignore rejection
});
```

Here's the point to remember about reaction functions: no matter how the code is formatted, only *one* of the two functions will ever be executed—never both. It's one or the other. In this regard, it's somewhat analogous to a try/catch block. It's also worth noting that the parameter of the reaction function is what the promise hands you back (with either the fulfilled value or the rejection reason). When that happens, you have your results.

Let's take a look at the `then()` function in action. The following listing updates your shopping cart and uses a promise instead of a callback function to process the results.

Listing 7.5 Processing a shopping cart update via a promise

The update returns a promise

```
CartDataSrc.updateCart(cartObj).$promise
.then(
   function(updatedCart) {
      console.log("Cart ID: " + updatedCart.id);
   },
   function(errorMsg) {
      console.log("Error: " + errorMsg);
   }
);
```

Use the promise's `then()` function to access the results

Having a promise returned is built into AngularJS's `$resource` methods. As you can see in the example, you're writing out the results of the call to the console as you did before—only this time you're able to use the returned promise object instead of diverting control over to a callback function. The `then()` method passes the success results or the rejected reason to the functions you give it.

Another perk of using promises is that you can chain multiple `then()` methods together if more than one thing needs to happen after your call has been made.

CHAINING PROMISES

Often after a process has run, you want several things to happen after the fact. In addition, you may need these things to happen in order, ensuring that the next event happens only if the one before it succeeds. This is not only possible but also easy to do with promises.

> **NOTE** jQuery's implementation of promises doesn't support every scenario described in this section. See https://blog.domenic.me/youre-missing-the-point-of-promises for more details.

So far in your shopping cart update, you've been printing the results to the console. In a real application, you want to perform the following tasks after the server call finishes:

1. Recalculate the cart's total, applying necessary discounts.
2. Update the view with the results.
3. Reuse the message service to update the user that the call was a success.

Moreover, you want these tasks performed in order, and only if each task is successful should the next one begin. This ensures that the user won't be erroneously notified that everything went swimmingly if an error happens to occur along the way (see the following listing).

Listing 7.6 Using promises to force control flow

`$resource` **returns a promise**

```
var promise = CartDataSrc.updateCart(cartObj).$promise

promise.then(function( updatedCart ) {
   return shoppingCartSvc
   .calculateTotalCartCosts(updatedCart);
})
.then(function(recalculatedCart) {
   replaceCartInView(recalculatedCart);
})
.then(function() {
   messageSvc.displayMsg("Cart updated!");
})
["catch"](function(errorResult) {
   messageSvc.displayError(errorResult);
});
```

Return recalculated cart for use in next `then()`

Display recalculated cart

Display a user message

Handle any errors that occurred along the way

This works because each `then()` returns a promise. If the reaction of the previous `then()` returns a promise, its value is used in the subsequent promise handed to the next `then()`. If the reaction returns a simple value, this value becomes the value in the promise passed forward. This allows you to chain them all together and makes for a straightforward and clean approach.

Being able to chain together multiple tasks in sequence in a few lines of code is amazing, but chaining can help you in other ways. Another amazing thing about chaining promises is that you can have more than one asynchronous process in the chain.

CHAINING MULTIPLE ASYNCHRONOUS PROCESSES IN SEQUENCE

Sometimes when you need several tasks to run in order, more than one may be asynchronous. Because you don't know when asynchronous processes will finish, trying to place one into a sequence with other tasks might be pretty challenging. It's easy, though, using promises. Because each `then()` is resolved before the next one is executed, the entire chain executes sequentially. This is still true even if multiple asynchronous processes are in the chain.

To demonstrate, let's pretend that the server APIs require you to use the cart ID that's returned by the shopping cart update in a subsequent `GET` call in order to properly display the cart onscreen. The following listing illustrates how to use promises to do this.

Listing 7.7 Executing more than one server call in order

```
var promise = CartDataSrc.updateCart(cartObj).$promise

promise.then(function( cartReturned ) {
   return shoppingCartSvc
   .getCartById(cartReturned.cartId);
})
.then(function(fetchedCart) {
   return shoppingCartSvc
   .calculateTotalCartCosts(fetchedCart);
})
.then(function(recalculatedCart) {
   replaceCartInView(recalculatedCart);
})
.then(function() {
   messageSvc.displayMsg(userMsg);
})
["catch"](function(errorResult) {
   messageSvc.displayError(errorResult);
});
```

Use the cart returned in the update for the next server call

Use the fetched cart for the recalculation

In this chain, your update happens first. Then, after it returns, your next server call fires. Because the `GET` call from `$resource` already creates a promise, its value will be used in the promise passed to the next `then()`.

Before finishing this discussion of promises, let's get a quick overview of error handling. You saw error handling in some of the examples, but I didn't go over any details.

7.3.4 Promise error handling

You can handle rejected promises in two ways. You saw the first way early on. Option 1 is to use the second reaction function of the promise's `then()` method. The second reaction is the one triggered when there's a rejection:

```
promise.then(
   function (value) {
   },
   function (reason) {
      // deal with the rejection
   }
);
```

Option 2 is to add an error-handling method called `catch()` to the end of your chain:

```
.catch(function (errorResult) {
    // deal with the rejection
});
```

Some browsers take issue with a method called `catch()`, because it's a preexisting term in the JavaScript language. Alternatively, you can use this syntax:

```
["catch"](function(errorResult) {
    // deal with the rejection
});
```

> **TIP** Writing `.catch()` as `["catch"]` looks strange but will help you avoid potential issues for any older browsers that don't support ECMAScript 5. If you use this syntax, as shown in these examples, notice that it *doesn't* have a dot in front of it.

You saw the second option being used with your shopping cart call. It used the message service to log the error and broadcast a user-friendly message to the user:

```
["catch"](function(errorResult) {
   messageSvc.displayError(errorResult);
});
```

With either method of error handling, rejections are passed down the chain to the first available error handler. This behavior seems obvious with the `catch()` method. What's less obvious is that this is true even when using the optional reaction function for error handling. If a rejection occurs somewhere up the chain, and *either* type of error-handling method is encountered somewhere down the chain (even if it's several `then()`s later), that error handler will be triggered and passed the error thrown.

As illustrated in our shopping cart examples, promises are powerful yet easy to use if you understand them. Frameworks and libraries sometimes add even more functionality, on top of what this chapter covers on promises. See their documentation for specific details.

Even if you have a project that requires you to support older browser versions, you can still use promises via your MV* framework if promises are supported or via a

third-party library. The following are a few of the many popular promise third-party libraries at the time of this writing:

- *bluebird*—https://github.com/petkaantonov/bluebird
- *Q*—https://github.com/kriskowal/q
- *RSVP.js*—https://github.com/tildeio/rsvp.js
- *when*—https://github.com/cujojs/when
- *WinJS*—http://msdn.microsoft.com/en-us/library/windows/apps/br211867.aspx

Aside from all of these being promise libraries, they also conform to the current preferred promise standard called *Promise/A+*. This is the same standard that native JavaScript promises are based on. If you'd like to read more about the Promise/A+ specification, a good resource is https://github.com/promises-aplus/promises-spec.

As an aside, jQuery also has its own version of promises, but as of this writing they aren't Promise/A+ compliant. With jQuery, promise functionality is done via its `Deferred` object. If you're interested, a great resource is the jQuery site itself: http://api.jquery.com/category/deferred-object.

Promises are also being implemented into the ECMAScript 6 (Harmony) version of JavaScript. Even before the specifications have been finalized, they already have limited support in many of today's browsers.

At this point, you're almost ready for our project. But you need to review the consumption of RESTful services first.

7.4 Consuming RESTful web services

This section covers consuming RESTful web services from your SPA. In many single-page applications today, these types of services are extremely common.

7.4.1 What is REST?

REST stands for *Representational State Transfer*. REST isn't a protocol or even a specification but an architectural style for distributed hypermedia systems. It has gained such widespread popularity that many MV* frameworks not only provide out-of-the-box support for it but also favor this style by default.

In a RESTful service, APIs define the media types that represent resources and drive application state. The URL and the HTTP method used in the API define the processing rules for a given media type. The HTTP method describes what's being done, and the URL uniquely identifies the resource affected by the action. REST can best be defined by describing its set of guiding principles.

7.4.2 REST principles

This section presents a few of the REST principles that most affect how you consume RESTful web services. This will also give you a good idea of what REST is about.

EVERYTHING IS A RESOURCE

One of the fundamental concepts in REST is that everything is a resource. A *resource* is represented with a type and conceptually maps to an entity or set of entities. A resource could be a document, an image, or information that represents an object such as a person. The notion of a resource could also extend to a service such as *today's weather* or, in our case, *a shopping cart*.

EVERY RESOURCE NEEDS A UNIQUE IDENTIFIER

Each resource in a RESTfull service should have a unique URL to identify it. This often entails creating and assigning unique IDs to the resource. You want to make sure that any ID you use in a URL in no way jeopardizes the security or integrity of your application. A common security measure is to assign a randomly generated ID for any resource that's personal or confidential. To ensure that the ID is used by only the intended user, the server-side code makes sure the requester is the authenticated user assigned to the resource and has the proper authorization to perform the action on the resource.

REST EMPHASIZES A UNIFORM INTERFACE BETWEEN COMPONENTS

You've already seen how HTTP methods are considered the verb of a web service call. Resource identifiers and HTTP methods are used to provide a uniform way of accessing resources. Table 7.2 gives some examples from the project.

Table 7.2 URLs in REST uniquely identify a resource, and the HTTP method describes that action being performed on the resource.

REST
URL: /shopping/carts/CART_ID_452 Method: `GET` Purpose: Fetch cart
URL: /shopping/carts/CART_ID_452/products/cod_adv_war Method: `POST` Purpose: Add an item to the cart
URL: /shopping/carts/CART_ID_452 Method: `PUT` Purpose: Update the entire cart's contents
URL: / shopping/carts/CART_ID_452/products/cod_adv_war Method: `DELETE` Purpose: Remove all instances of a particular product from the cart

It's important to note that the style of URL used isn't part of REST, even though you sometimes see the phrase *RESTful URL* used in articles about REST.

INTERACTIONS ARE STATELESS

Session state for your application should be held in your SPA and shouldn't rely on client context being stored on the server between requests. Each request made by the

SPA to the server should convey all the information needed to fulfill the request and allow the SPA to transition to a new state.

Again, we've barely scratched the surface of REST here. For more information about REST and REST architecture, see http://en.wikipedia.org/wiki/Representational_state_transfer.

7.4.3 How MV* frameworks help us be RESTful

Thinking in terms of REST can take a little getting used to. Fortunately, MV* frameworks such as Backbone.js and AngularJS support REST right out of the box. For example, when you used Backbone.js for your shopping cart update, it automatically added the ID from your model to your URL so that the URL uniquely identifies the resource in the request. Frameworks that don't have explicit models, such as AngularJS, might allow you to use path variables in a URL template to create a RESTful URL. You'll see examples of path variables in a moment, when you look how AngularJS's `$resource` object is used in your project.

MV* frameworks also help you consume RESTful services by making it easy to send the correct HTTP request method. They usually either come with canned functions for `GET`, `POST`, `PUT`, and `DELETE` or allow you to effortlessly generate them via configuration.

Now that you have a general idea of RESTful services and their guiding principles, you're finally ready to tackle the project. In this project, you'll get to see firsthand how promises and REST work together to maintain a shopping cart.

7.5 Project details

You'll continue building on the preceding chapter's used video game project by adding a shopping cart. As usual, you'll use AngularJS for your MV* framework. It has built-in support for both promises and the consumption of RESTful services. As indicated at the beginning of this chapter, we discuss the server side of the application only conceptually here.

Because many server-side languages and frameworks might be used instead of what you're using, I include a small summary of the tasks the server will need to perform for each call in appendix C. This way, you can create the server-side code by using a different tech stack if you wish. As always, the complete code is available for download. Let's begin by walking through the setup of your data source.

7.5.1 Configuring REST calls

Earlier in this chapter, you learned about the `$resource` object from AngularJS. It makes consuming RESTful services easier, and its methods all return promises. You'll use it for every server call in your project. Although I try to keep our discussions framework neutral, you'll have to take a moment to further review how `$resource` works. It can be a little intimidating at first. After you walk through how it works, though, you'll see how easy it is to use. After a gentle introduction to `$resource`, you'll proceed with how it'll be configured in your example SPA's shopping cart service.

Creating URLs with AngularJS's $resource

Like some of the other MV* frameworks, AngularJS offers support for RESTful service web service consumption out of the box by using its `$resource` object. This object adds a lot of sugar coating for the underlying `XMLHttpRequest` object to hide much of the boilerplate code you'd have to otherwise write yourself.

The main goal of `$resource` is to make it easy to work with RESTful services. Having a consistent and uniform way to represent resources is one of the principles of REST. After a URL style has been established, the `$resource` factory will help you create URLs that conform to this style easily.

The `$resource` factory enables you to define a template that will create resource URLs for each type of REST call you need to make. To use `$resource`, you can pass a URL, optional default parameters, and an optional set of actions to its constructor:

```
$resource(DEFAULT URL, DEFAULT URL PARAMS, OPTIONAL ACTIONS)
```

The following will serve as your default URL:

```
"controllers/shopping/carts"
```

The default will be used if you don't override it. But in this project, you're defining custom functions that will override it with their own URLs. Each custom action can have its own. To construct the URLs in the structure needed by your RESTful web services, you can use URL path parameters. As with routes, using a colon in front of a string in the URL indicates a parameter. Here's an example URL from your configuration that includes URL path parameters:

```
"controllers/shopping/carts/:cartId/products/:productId"
```

The next argument, the optional parameter list, acts like a data map. It tells the `$resource` object that in one or more of these calls, this optional parameter list *may* be used. This list is in the form of key-value pairs. The left side is the name of a parameter in the URL. The right side is the value for the parameter. The `@` symbol tells `$resource` that the value is a *data property name*, not just a string. With it present, the data object passed in will be scanned for a property with that name, and its value will be used in the URL's path.

```
{
   cartId : "@cartId",
   productId : "@productId "
}
```

For example, if you passed in an object called `myCart` for the call, then the value for the URL parameter `cartId` would come from `myCart.cartId`. The value for the URL parameter `productId` would come from `myCart.productId`.

The nice thing about using `$resource` as a REST URL template is that you get a set of REST calls out of the box that are preconfigured with the following HTTP methods:

`get()`—`GET`

`query()`—`GET` (intended for a list; by default it expects an array)

`save()`—`POST`

(continued)

`delete()`—`DELETE`

`remove()`—`DELETE` (identical to `delete()`, in case the browser has a problem with the `delete()` action)

If you want to customize your calls as we're doing, you can pass in the optional set of named functions (or *actions* in Angular-speak). You can use the action to create a completely customized call or override one of the out-of-the-box functions. For example, to create a custom action called `updateCart()`, you can include the following in your set of actions:

```
updateCart : {
   method : "PUT",
   url : "controllers/shopping/carts/:cartId"
}
```

After you have the `$resource` object configured, any calls you make with it automatically return a promise. You've already seen how to use them to work with the results of your calls.

In this chapter's examples, you're using `$resource` inside your shopping cart service because you have additional processes taking place before the data is returned to the controller. For simple data returns, you might want to wrap the `$resource` in another AngularJS object (such as a factory) and include it directly in your controller.

To see the complete documentation for `$resource`, visit the AngularJS site at https://docs.angularjs.org/api/ngResource/service/$resource.

Now that you've looked at `$resource` basics, let's look at the entire code for the `$resource` instance used for your shopping cart (see the following listing). This will give you a picture of the type of calls that will be made inside the shopping cart service.

Listing 7.8 Configuration for your REST calls

```
var Cart = $resource("controllers/shopping/carts", {
   cartId : "@cartId",
   productId : "@productId"
}, {
   // cart methods
   getCart : {
      method : "GET",
      url : "controllers/shopping/carts/:cartId"
   },
   updateCart : {
      method : "PUT",
      url : "controllers/shopping/carts/:cartId"
   },
   // item-level methods
   addProductItem : {
      method : "POST",
      url : "controllers/shopping/carts/:cartId/products/:productId"
```

Assign the $resource created to a variable

Define default parameters

Define actions for the rest of your calls

```
    },
    removeAllProductItems : {
       method : "DELETE",
       url : "controllers/shopping/carts/:cartId/products/:productId"
    },
});
```

With `getCart()`, you can get the cart's content anytime you need it. You'll use `addProductItem()` to add a new product to the cart or use `removeAllProductItems()` to remove all quantities of a given product type. You can use `updateCart()` to update the entire cart.

> **TIP** Though you're not implementing security in this application, usually each call you make is validated for security and data integrity in the server-side code.

Because the previous chapter covered the application, in this section you'll focus only on the code around your server calls and how to process the results. Let's begin with adding new product items to the cart.

7.5.2 Adding product items to the cart

Following the URL format chosen earlier, you include the cart ID and the product ID in your RESTful service call to add a product item to the shopping cart (see table 7.3). If the product already exists in the cart, the quantity increases.

Table 7.3 RESTful call to add a product to the shopping cart

Method	URL	HTTP method	Request	Response
`Cart.addProductItem()`	`/shopping /carts/ CART_ID_89/product s/ cod_adv_war`	`POST`	`Products`	`Cart`

You've modified the product display page to include a new button that will make the call to add a product item to the cart. When this button is clicked, it calls the `addItem()` action in your shopping cart's `$resource`'s configuration. The following listing shows the modified view containing the new button.

Listing 7.9 Updated product display view

```
angular.module("data.appData", [])

<section class="product_info">
   <h2>
      {{results.name}}

      <span class="price">
        Used Price:
```

```
        {{results.discountPrice | currency:"$":0}}
      </span>

      <button id="product_info_add_btn"
        ng-click="addToCart('{{results.productId}}')">
        Add to Cart
      </button>
    </h2>

    <section id="product_info_img_container">
      <img id="product_info_img"  ng-src="{{results.url}}">
    </section>

    <section id="product_info_summary">
      {{results.summary}}
    </section>
</section>
```

Pass the product ID of the game found

Figure 7.6 shows what the finished view looks like.

Using an `ng-click` binding, you've bound the button click to a function called `addToCart()` on the `$scope` (ViewModel) in the controller to handle the new user action. In turn, this function calls the `addToCart()` function of your shopping cart service (see listing 7.10). As a reminder, the AngularJS `$stateParams` object allows you to access parameters from the route that was executed.

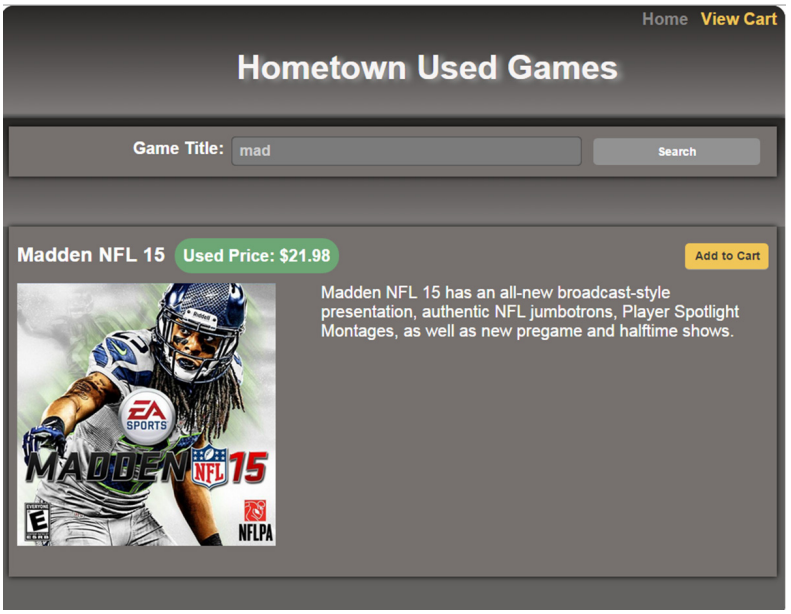


Figure 7.6 The product display page now features a button to add the item to the shopping cart.

Listing 7.10 Application's data holds cart ID

```
$scope.addToCart = function() {
   shoppingCartSvc.addToCart($stateParams.productId);
};
```

Use the shopping cart service to add the item

After the function call is made, the `addToCart()` function in the shopping cart service makes the RESTful call to the server for processing (see the following listing).

Listing 7.11 Function to make the `addItem()` call

```
function addToCart(productId) {
   return Cart.addProductItem({
      cartId : createorGetExistingCart(),
      productId : productId
   }).$promise.then(function(cartReturned) {

      messageSvc.displayMsg("Item added to cart!");

      console.log("Item added successfully to cart ID "
      + cartReturned.cartId);
   })

   ["catch"](function(error) {
      messageSvc.displayError(error);
   });
}
```

Pass the cart ID and the product ID as call parameters

Promise chain: display user message

Handle any errors

The product ID and the cart ID get mapped to the default parameters of your `addProductItem()` custom action in the `$resource` configuration that you saw earlier. After the user has added items to the cart, a new view needs to display the cart's contents. For this, you've added a brand-new view to the application.

7.5.3 Viewing the cart

In this call, you use the cart ID that was generated locally when the user landed on the welcome page. You can use it to get the current state of the cart. Table 7.4 lists this call's properties.

Table 7.4 RESTful call to get the shopping cart to display its contents in the view

Method	URL	HTTP method	Request	Response
`Cart.getCart()`	`/shopping /carts/CART_ID_89`	`GET`	`Empty`	`Cart`

To be able to view the cart from anywhere, a new link is added to the header. Clicking the link executes the `viewCart` route, which takes you to the shopping cart view:

```
<a id="viewCartLink" ui-sref="viewCart">View Cart</a>
```

When the controller behind the shopping cart view is called, the first thing it does is make a `GET` call to retrieve the cart from the server. You'll look at this call from the controller first and then the shopping cart service.

In the controller where the call originates, the `getCart()` function returns the promise generated by the `$resource` call. As you may remember from our discussion of promises, the promise referenced here will be pending until the call completes:

```
var promise = shoppingCartSvc.getCart();
handleResponse(promise, null);
```

You're also handing off the promise to a generic JavaScript function in the shopping cart controller that will handle the promise returned. The nice thing about promises is that they can be passed around like any other JavaScript object. In each call, whether it's fetching the cart, updating it, or removing an item, you'll process the promise in the same way every time (see the next listing).

Listing 7.12 Generic function to handle all cart promises

```
function handleResponse(promise, userMsg){

   promise.then(function( cartReturned ) {  B
      return shoppingCartSvc
      .calculateTotalCartCosts(cartReturned);
   })

   .then(function(recalculatedCart) {
      replaceCartInView(recalculatedCart);
   })

   .then(function(recalculatedCart) {
      messageSvc.displayMsg(userMsg);
   })

   ["catch"](function(errorResult) {
      messageSvc.displayError(errorResult);
   });
};
```

Promise and optional user message passed in

Recalculate cart

Update the view

Display user message

Handle any errors

In the shopping cart service, your call to get the cart becomes a one-liner thanks to the magic of the out-of-the-box support for REST in your MV* framework. Here you're passing an object with the ID of your cart as the payload of your call. The object will be scanned by `$resource` for a property name that matches the `cartId` URL parameter. Because you've stored the cart ID in the `cartData` object in the client, you can use it when you need the ID in the URL:

```
function getCart() {
  return Cart.getCart({cartId : cartData.cartId})
  .$promise;
}
```

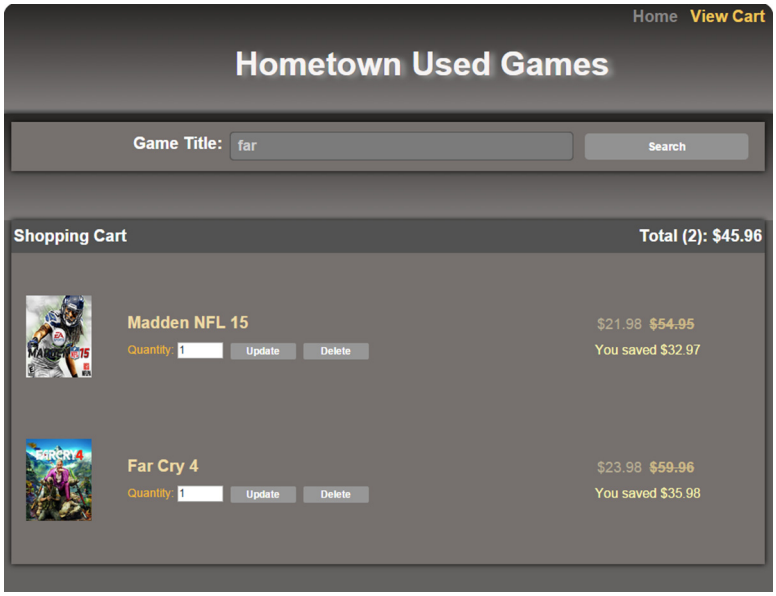


Figure 7.7 The shopping cart view allows the user to modify the cart's contents.

Also remember that `Cart` is the variable name assigned to the `$resource` object you created. When the `Cart.getCart()` call completes, the promise is returned to the controller for the processing you saw previously. If all the promises are fulfilled in the promise chain when the call completes, the view displays all the items currently in the cart. It also shows the original price of each item, its used price, as well as the cost savings. At the top of the cart is a running total of all items and their used prices (see figure 7.7).

With your cart returned, the user can use the UI controls to update it or delete items from it.

7.5.4 Updating the cart

When you update the cart, you're not sending only the new items; you're sending and receiving the entire cart. The RESTful URL identifies the cart you're updating, and the request body has the updated cart data. Table 7.5 has this call's properties.

Table 7.5 RESTful call to update the shopping cart with new input from the user

Method	URL	HTTP method	Request	Response
`Cart.updateCart()`	`/shopping /carts/CART_ID_89`	`PUT`	`Cart`	`Cart`

For each entry, you provide an input control to let the user enter a new item count. You also have a button that will update the entire cart by each item. Each update

button updates the entire cart in the same manner. It's repeated beside each item only for convenience.

```
<span class="cartQuantityLabel">Quantity: </span>

<input type="text" ng-model="game.quantity" size="4">

<button class="cartItemButton" ng-click="updateQuantity()">
   Update
</button>
```

The `updateQuantity()` function needs no parameters because it always passes the entire cart. In the controller, you rely on the shopping cart service to make the update and pass the promise returned to your generic promise handler (see the following listing).

Listing 7.13 Controller code for cart updates

```
$scope.updateQuantity = function() {
   var uCart =
   shoppingCartSvc.createCartForUpdate($scope.cart);

   var promise = shoppingCartSvc.updateCart(uCart);

   handleResponse(promise, "Cart updated!");
};
```

Create a request object for the request body

Make the update call, return the promise

Pass the promise and user message to the generic handler

In the shopping cart service, you have a JavaScript function to create a cart object to send to the server. To make the request leaner, in the next listing you include only IDs and updatable properties.

Listing 7.14 Building the update request object

```
function createCartForUpdate(cartFromView) {

   var cart = {
      cartId : cartFromView.cartId,
      totalCount : cartFromView.totalCount,
      items : new Array()
   };

   angular.forEach(cartFromView.items, function(item) {
      var pItem = {
         productId : item.productId,
         quantity : item.quantity
      };
      cart.items.push(pItem);
   });

   return cart;
};
```

Create the object with a placeholder for the items array

Iterate over the cart items and add them to the request

When the request object is ready, you can make the update request. Again, thanks to our MV* framework's support for REST, you have a one-liner:

```
function updateCart(cart) {
   return Cart.updateCart(cart).$promise;
};
```

Like the other call, the update returns the promise to the controller so the promise chaining can process the results.

The last thing you need to do in the cart is provide the ability to remove items from it. In the next section, you'll examine how to remove all quantities of a particular product type from the cart.

7.5.5 Removing products from the cart

To remove all items of a product from the cart, the most obvious choice in HTTP methods is `DELETE`. You need to make sure that you're identifying both the cart and the product, just as you did when you added it. Table 7.6 has this call's properties.

Table 7.6 RESTful call to remove all items of a product from the shopping cart

Method	URL	HTTP method	Request	Response
`Cart.removeAllProductItems()`	`/shopping/carts/CART_ID_89/products/cod_adv_war`	`DELETE`	`Empty`	`Cart`

In addition to users having the ability to update the quantity, the Delete button next to each product enables users to remove it completely. In the view, you've bound the button's click to the `removeItem()` function on the controller. The function call passes forward the product ID of the product that's being deleted.

```
<button class="cartItemButton"
ng-click="removeItem(game.productId)">Delete</button>
```

In the controller, as with getting the cart or updating it, you make the call and pass the returned promise to the generic promise handler (see the following listing).

Listing 7.15 Controller code for cart deletes

```
$scope.removeItem = function(productId) {
   var promise =
   shoppingCartSvc.removeAllProductItems(productId);

   handleResponse(promise, "Cart removed!");
};
```

Pass the product ID, get a promise back

Pass the promise and user message to the generic handler

Finally, you get to the matching code in the shopping cart service where the call is made (see the following listing). The cart's ID from your cart data object is mapped to

the `cartId` URL parameter, and the product ID passed in is mapped to the `productId` parameter.

Listing 7.16 Controller code for cart deletes

```
function deleteItem(productId) {
   return Cart.removeItem(

   {
      cartId : cartData.cartId,
      productId : productId
   }

   ).$promise;
};
```

Pass the cart ID and product ID to the call

Return a promise to the controller

Don't forget that if you want to create the project in your own environment, the server-side supplement in appendix C begins with a summary of the objects and tasks. This is included in case you're using a different tech stack than the example's code. Also, as usual, the complete source code is available for download.

7.6 Chapter challenge

Now here's a challenge for you to see what you've learned in this chapter. In the preceding chapter's challenge, you created a movie search. You displayed movie titles that matched wholly or partially the text the user typed into an input field. A key-up event was bound to a function that published the field's contents with each keystroke. A search module subscribed to that topic and performed a search accordingly. The search also used pub/sub to publish the results, which were displayed in an unordered list below the input field.

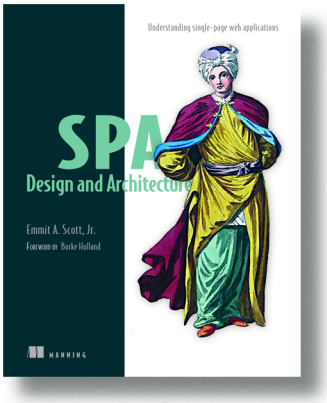
Extend this exercise by putting the stub data and the search logic on the server. You'll still have a client-side module listening for the input field contents to be published. In turn, it will fire the server call every time it hears the topic. On the server, you can use any technologies you're comfortable with. Make the server call a RESTful service call. Use a RESTful URL and an appropriate HTTP request method for this type of request. Use a promise to process the server call. Upon success, publish the results of the search. Write any errors to the console.

7.7 Summary

We covered a lot of ground in this chapter. Let's review:

- The server is still important to the single-page application, providing features such as security, validation, and standard APIs for accessing back-end data.
- Having an agreement between the client and the server on the data type and the HTTP methods is essential for a call to be successful.
- Native objects on both ends are converted to the agreed-upon data format in both the request and the response.

- MV* frameworks that support server communication often take care of routing tasks, such as providing standard request types out of the box and handling data conversions.
- MV* frameworks typically support server communication either through extending a parent model or via a data source object.
- Call results are handled either through callbacks using continuation-passing style or through promises, depending on the framework or library used.
- A promise represents the outcome of a pending asynchronous process. It starts as pending but eventually transitions to either fulfilled or rejected when the call completes.
- A promise has several methods, but the most commonly used one is `then()`. This method allows you to register two functions (called *reactions*) to process a fulfilled or rejected promise.
- The reaction for a fulfilled promise gives you access to the result data of a process. The reaction for a rejected promise contains the rejected reason (usually an `Error` object).
- Reactions can be chained together to control the flow of a group of processes, even if other asynchronous calls are in the chain.
- Promise errors can be handled either through the rejection reaction or via a `catch()` method.
- *REST* stands for *Representational State Transfer* and is an architectural style for developing web services.
- In REST, everything is a resource and should have a unique URL representing it.
- HTTP methods describe the action for the resource. The four most common are `GET`, `POST`, `PUT`, and `DELETE`.



The next step in the development of web-based software, single-page web applications deliver the sleekness and fluidity of a native desktop application in a browser. If you're ready to make the leap from traditional web applications to SPAs, but don't know where to begin, this book will get you going.

SPA Design and Architecture teaches you the design and development skills you need to create SPAs. You'll start with an introduction to the SPA model and see how it builds on the standard approach using linked pages. The author guides you through the practical issues of building an SPA, including an overview of MV* frameworks, unit testing, routing, layout management, data access, pub/sub, and client-side task automation. This book is full of easy-to-follow examples you can apply to the library or framework of your choice.

What's inside

- Working with modular JavaScript
- Understanding MV* frameworks
- Layout management
- Client-side task automation
- Testing SPAs

This book assumes you are a web developer and know JavaScript basics.

Sharing and Securing Web Things

Gartner research predicts that there will be tens of billion web-connected devices deployed by 2020, creating a complex network of APIs, communication protocols, and application designs. This chapter introduces API security in the world of connected devices, using several techniques and protocols including OAuth.

Chapter 9 from *Building the Web of Things*
by Dominique D. Guinard and Vlad M. Trifa

Share: Securing and sharing web Things

This chapter covers

- A short overview of security risks and issues on the Web of Things
- A brief theoretical introduction to HTTPS, certificates, and encryption
- Best practices and techniques for web-based authorization and access control
- Learning to implement these best practices and tools on your Raspberry Pi
- Implementing the Social Web of Things in the WoT gateway

In most cases, Internet of Things deployments involve a group of devices that communicate with each other or with various applications within closed networks—rarely over open networks such as the internet. It would be fair to call such deployments the "intranets of Things" because they're essentially isolated, private networks that only a few entities can access. But the real power of the Web of Things lies in opening up these lonely silos and facilitating interconnection between devices and applications at a large scale.

Why would you even want this? When it comes to a critical IoT system such as a network of industrial machines in a large factory in Shenzhen, the security system of the British Museum, or simply own collection of smart devices at home, you certainly don't want these networks to be open to anyone. But when it comes to public data such as data.gov initiatives, real-time traffic/weather/pollution conditions in a city, or a group of sensors deployed in a jungle or a volcano, it would be great to ensure that the general public or researchers anywhere in the world could access that data. This would enable anyone to create new innovative applications with it and possibly generate substantial economic, environmental, and social value. Another use case is the smart hotel scenario presented in chapter 1, where hotel guests (and only *they*) should have access to some services and devices in their room (and only *there*) during their stay (and only *then*). Because the public infrastructure is becoming not only digital but also pervasive, the earlier we could build, deploy, and scale those systems while maximizing the ability to share data between devices, users, and applications, the better it would be for all of us. How to share this data in secure and flexible way is what Layer 3 provides, as shown in figure 9.1.

The prerequisite for this is to use a common protocol and data format between devices and applications, which we covered extensively in the previous chapters. But once devices are connected to a public network, the most important problem to solve is how to ensure that only a specific set of users can access only a specific set of resources at a specific time and in a specific manner. In the next sections we'll show how to do this by building on the concepts and tools you've already seen. First, we'll

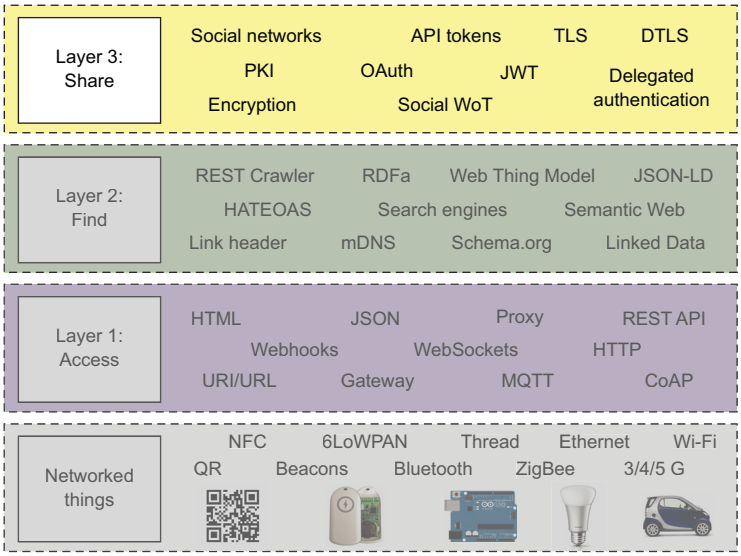


Figure 9.1 The Share layer of the Web of Things. This layer focuses on how devices and their resources must be secured so that they can only be accessed by authorized users and applications.

show how Layer 3 of the WoT architecture covers the security of Things: how to ensure that only authorized parties can access a given resource. Then we'll show how to use existing trusted systems to allow sharing physical resources via the web.

9.1 Securing Things

Right now, the hottest topic (or potato, to be more accurate!) of the IoT world is arguably security.[1] We keep hearing all over the news about the IoT and the endless possibilities of an all-connected world. Sadly, this vision is also continuously tainted by major security breaches: personal information, credit card data, sensitive documents, or passwords from millions of users being stolen by hackers. Such happenings not only can severely hurt the reputation of a company but also can have disastrous effects for the users. Ultimately, every security breach hurts the entire web because it erodes the overall trust of users in technology. No one wants their smart fridges sending spam emails[2] about dubious pills, inheritances, or unclaimed lottery gains.

Security in the Web of Things is even more critical than in the web. Because web Things are physical objects that will be deployed everywhere in the real world, the risks associated with IoT attacks can be catastrophic. Digitally augmented devices allow collecting much more information about people with a fine-grained resolution, such as when you got your last insulin shot, what time you go jogging and where, and the like. But more important, unauthorized access to physical objects can be dangerous—remotely controlling your brand new BMW[3] or house,[4] anyone? Despite those risks, recent reports have shown a sad state of affairs in the world of IoT security.[5] Although many vulnerabilities—called *exploits* in hacker parlance—are widely known and patches for them are readily available, it has been reported that the majority of IoT solutions don't comply with even the most basic security best practices; think clear-text passwords and communications, invalid certificates, old software versions with exploitable bugs, and so on. In other words, you don't even have to be a security expert to use existing weaknesses in many services or devices and gain access to unauthorized content.

This book is not about network security, so don't expect to become an expert in this field by the end of this chapter. But because it's such a crucial issue for any production system or consumer product connected to the internet, we'll cover the basics you need to know when building IoT solutions in the form of a set of best practices for building secure and reliable devices and applications. If you can't wait any longer, an excellent resource is the Open Web Application Security Project (OWASP) Internet of Things project,[6] which contains useful, down-to-earth, and practical information about how to build safer IoT applications and systems.

---

1 http://venturebeat.com/2016/01/16/ces-2016-the-largest-collection-of-insecure-devices-in-the-world
2 http://www.theguardian.com/technology/2014/jan/21/fridge-spam-security-phishing-campaign
3 http://www.wired.com/2015/08/bmw-benz-also-vulnerable-gm-onstar-hack/
4 http://www.forbes.com/sites/kashmirhill/2013/07/26/smart-homes-hack/
5 See "Insecurity in the Internet of Things," https://www.symantec.com/iot/.
6 https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project

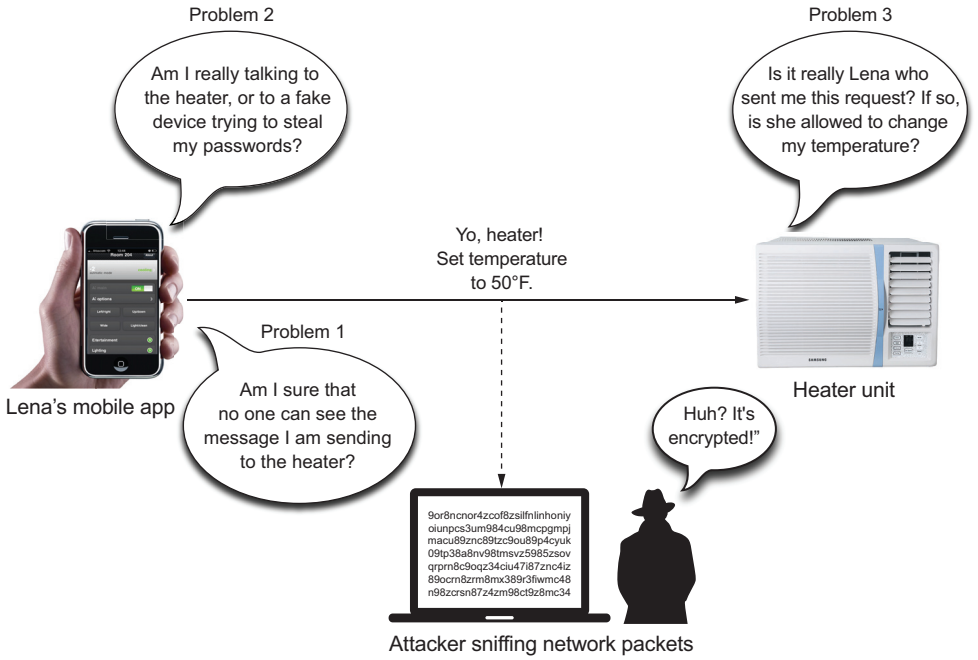


Figure 9.2 The three principal challenges in securing the IoT. First, communications must be encrypted to prevent unauthorized entities from reading the messages between a client and a server. Second, the client must be sure they are really talking to whom they they are. Third, the server must be sure that a message comes from an authorized client allowed to send that request.

Roughly speaking, securing the IoT comes down to solving three major problems summarized in figure 9.2:

- First, we must consider how to encrypt the communications between two entities (for example, between an app and a web Thing) so that a malicious interceptor—a "man in the middle"—can't access the data being transmitted in clear text. This is referred to as securing the channel and will be covered in section 9.1.1.
- Second, we must find a way to ensure that when a client talks to a host, it can ensure that the host is really "himself," which is the topic discussed in section 9.1.2. For example, in chapter 4, you downloaded and installed NOOBS from our website. But you did so via HTTP instead of HTTPS, and we didn't provide an SHA checksum available for that image over HTTPS. In essence, you had to trust that whatever you downloaded was really what we gave you and not a corrupted OS image inserted by an attacker.
- Third, we must ensure that the correct access control is in place. We need to set up a method to control which user can access what resource of what server or Thing and when and then to ensure that the user is really who they claim to be. This topic will be covered in section 9.2.

After exploring these three problems and their solutions, in section 9.3 we'll blur another line: the one between the Social Web and the Web of Things. We'll put together everything you've learned so far to build an application that allows you to use third-party social network identities to share Things with your friends.

9.1.1 Encryption 101

As you've seen before, there's more to security than encryption. Nevertheless, encryption is an essential ingredient for any secure system. Without encryption, any attempt to secure a Thing will be in vain because attackers can sniff the communication and understand the security mechanisms that were put in place.

Using a web protocol without encryption can be compared to sending a postcard via snail mail: anyone can read the content of the postcard at any stage. Adding encryption to a web protocol is like putting the postcard in a thick and sealed envelope: even if you can see the envelope, you can't read the card!

SYMMETRIC ENCRYPTION

The oldest form of encoding a message is *symmetric encryption*. The idea is that the sender and receiver share a secret key that can be used to both encode and decode a message in a specific way; for example, by substituting or shifting some characters by a number. This type of encryption is the easiest to put in place for resource-limited IoT devices, but the problem is that as soon someone discovers the key, they can decode and encode any message. To use a symmetric key successfully, the key has to be shared with trusted parties securely, such as by giving the key to the recipient in person.

ASYMMETRIC ENCRYPTION

In the internet era, another method called *asymmetric encryption* has become popular because it doesn't require a secret to be shared between parties. This method uses two related keys, one public and the other private (secret), as shown in figure 9.3. A host

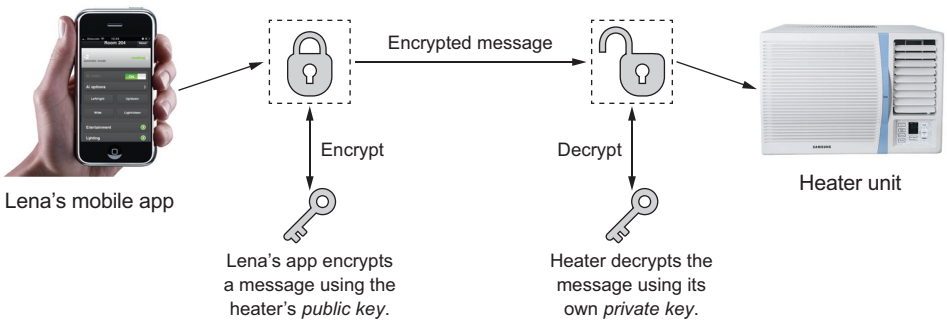


Figure 9.3 Asymmetric encryption in an IoT context. The heater shares its public key with Lena. It's then up to Lena's mobile app to encrypt messages sent to the heater. Thanks to the power of cryptography, the only way to decrypt the message is with the private key of the heater.

can freely share its public key with anyone over the internet. When any client wants to send a message to the host, it can use the public key to encode the message before sending it. Once a message is encoded with the public key, it can be decoded only with the private key that's known only by the host. This way, any message sent by a client (for example, a web app) to a server (for example, a web Thing) can be opened only by the server and not by an eavesdropper.

9.1.2 Web security with TLS: the S of HTTPS!

Fortunately , there are standard protocols for securely encrypting data between clients and servers on the web. The best known protocol for this is Secure Sockets Layer (SSL). SSL has long been the technology that sits behind the *S* in HTTPS, which is the method used to encrypt all the communications between your browser and a web server. But a number of important vulnerabilities in the SSL protocol have been discovered over the years, making it possible for attackers to crack the security SSL provides. In 2014, major vulnerabilities in the SSL 3.0 protocols were found; for example, POODLE,[1] Heartbleed,[2] and Shellshock.[3] These events inked the death of this protocol, which was replaced by the much more secure but conceptually similar Transport Layer Security (TLS).[4]

This highlights two important points. First, no method or system is secure forever. Second, open protocols—and especially web protocols—are closely monitored and fixed as soon as flaws are identified. In consequence, all communications over the Web of Things are to be encrypted with TLS. We won't give a full description of TLS here because it would take a chapter on its own—or a whole book, for that matter—but we'll review the basics of TLS and focus on the key concepts while simplifying the complex bits.

TLS 101

Despite its name, TLS is an Application layer protocol (see chapter 5). TLS not only secures HTTP (HTTPS) communication but is also the basis of secure WebSocket (WSS) and secure MQTT (MQTTS). TLS has two main roles. First, it helps the client ensure that the server is who it says it is; this is the SSL/TLS authentication. Second, it guarantees that the data sent over the communication channel can't be read by anyone other than the client and the server involved in the transaction (also known as SSL/TLS encryption).

---

1 https://blog.mozilla.org/security/2014/10/14/the-poodle-attack-and-the-end-of-ssl-3-0/
2 https://en.wikipedia.org/wiki/Heartbleed
3 https://en.wikipedia.org/wiki/Shellshock_(software_bug)
4 The long legacy of SSL means that today the acronym SSL is used as an umbrella term for both TLS and SSL.

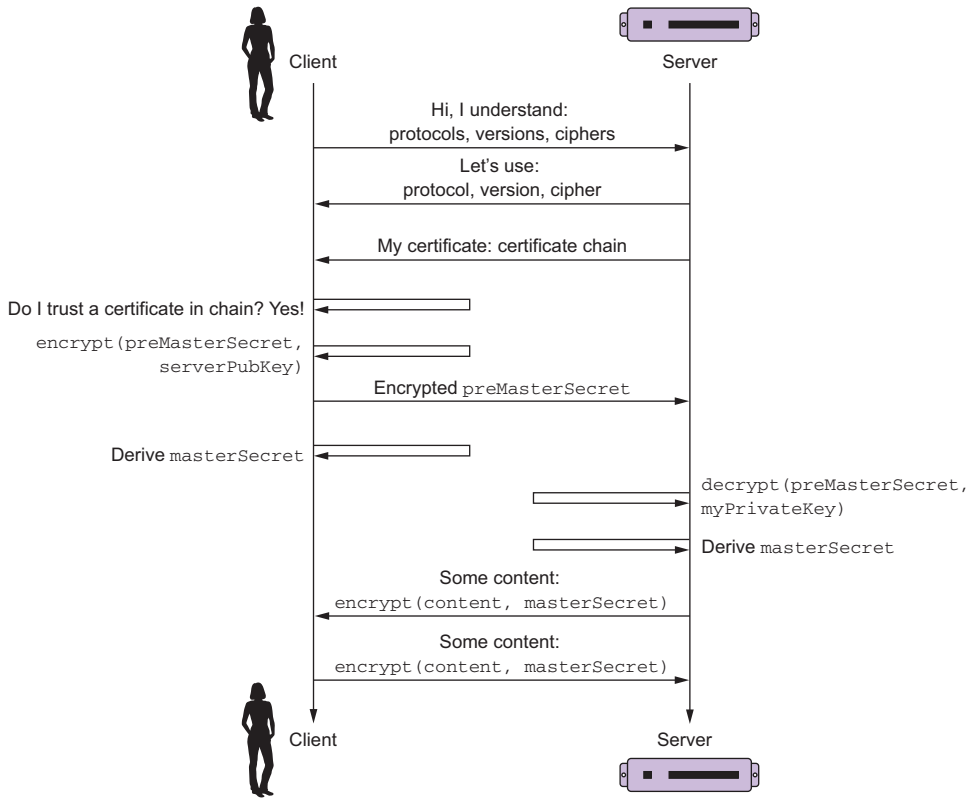


Figure 9.4 SSL/TLS handshake: the client and the server first negotiate the protocols and encryption algorithms, and then the server sends its certificate chain to prove who it is to the client. Finally, the client sends a preMasterSecret from which the client and server derive a masterSecret used to encrypt all the future messages.

A typical TLS exchange between a client and a server is shown in figure 9.4.[1] This is what happens when you use your browser to connect to an HTTPS website, such as https://manning.com. Here's a summary of the most important steps:

1. The client, such as a mobile app, tells the server, such as a web Thing, which protocols and encryption algorithms it supports. This is somewhat similar to the content negotiation process we described in chapter 6.
2. The server sends the public part of its certificate to the client. The goal here is for the client to make sure it knows who the server is. All web clients have a list of certificates they trust.[2] In the case of your Pi, you can find them in

[1] If you want an even simpler way of explaining TLS to your cat, check "What's Behind the Padlock": https://casecurity.org/wp-content/uploads/2013/01/ssl-1200.jpg.

[2] Firefox and Chrome, for example, trust certificates signed by those CAs; see https://mozillacaprogram.secure.force.com/CA/IncludedCACertificateReport.

/etc/ssl/certs. SSL certificates form a trust chain, meaning that if a client doesn't trust certificate S1 that the server sends back, but it trusts certificate S2 that was used to sign S1, the web client can accept S1 as well.

3 The rest of the process generates a key from the public certificates. This key is then used to encrypt the data going back and forth between the server and the client in a secure manner. Because this process is dynamic, only the client and the server know how to decrypt the data they exchange during this session. This means the data is now securely encrypted: if an attacker manages to capture data packets, they will remain meaningless.

9.1.3 Enabling HTTPS and WSS with TLS on your Pi

Now that you've seen the theory, it's time for a bit of practice! Let's secure the API of your WoT Pi to ensure that traffic between the Pi and its clients is encrypted. Note that the process we define here works as well on all the other Linux devices we talked about—for example, the Intel Edison or the BeagleBone—as well as on any Linux- or Unix-based machines. Go ahead and generate a certificate. First, you need to make sure the OpenSSL library is installed. On your Pi go to the /resources directory and run

```
sudo apt-get install openssl
```

This should tell you something along the lines of `openssl is already the newest version.` Or it will be installed if not present. Now, to generate the certificates, run

```
openssl req -sha256 -newkey rsa:2048 -keyout privateKey.pem -out caCert.pem
    -days 1095 -x509
```

Because this command is self-explanatory we won't detail it. No? Fine, let's dig into it! The command does two things in one. First, it generates a private key (`-newkey rsa:2048 -keyout privateKey.pem`) that will be used to sign the certificate using the sha256 hashing algorithm. While it does this, you'll see a `Generating a 2048 bit RSA private key` message followed by a prompt to provide a passphrase, essentially a password to protect your private key. Make sure you keep this one safe because you'll need it soon!

Second, it will generate a new certificate (`-out caCert.pem`) that will last for 1,095 days using the x509 data format, and it also prompts you with a few questions, as shown in listing 9.1. The common name is the hostname for which this certificate should be valid; for example, `raspberrypi.local` if you're on your Pi or `localhost` if you're running these examples on your machine. The information you provide here will be exposed in the certificate and will be visible to all clients.

Listing 9.1 Information requested when generating a self-signed certificate

```
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
```

```
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]: UK
State or Province Name (full name) [Some-State]: London
Locality Name (eg, city) []:London
Organization Name (eg, company) [Internet Widgits Pty Ltd]: Web of Things
Organizational Unit Name (eg, section) []: Web of Things
Common Name (e.g. server FQDN or YOUR name) []: raspberrypi.local
Email Address []:book@webofthings.io
```

This should be the hostname, IP, or domain name corresponding to your Pi or the local machine you test the code from.

At the end of this process, two files will be generated:

caCert.pem is the public part of the certificate your Pi server will send to the clients when connecting to it via TLS.

privateKey.pem is the private key of your Pi server and hence should be kept, well...private.

You're now ready to turn your Pi unencrypted HTTP and WS APIs into secure HTTPS and WSS APIs. All you need to do is modify the code of the wot-server.js file at the root of your WoT PI project (see chapters 7 and 8). Copy the content of wot-server.js into a new wot-server-secure.js file and modify it as shown in the following listing, which enables HTTPS and WSS.

Listing 9.2 Modifying the WoT Pi server to serve HTTPS and WSS content

```
[...]
var createServer = function (port, secure) {
  if (process.env.PORT) port = process.env.PORT;
  else if (port === undefined) port = resources.customFields.port;
  if (secure === undefined) secure = resources.customFields.secure;

  initPlugins();

  if(secure) {
    var https = require('https');
    var certFile = './resources/change_me_caCert.pem';
    var keyFile = './resources/change_me_privateKey.pem';
    var passphrase = 'webofthings';

    var config = {
      cert: fs.readFileSync(certFile),
      key: fs.readFileSync(keyFile),
      passphrase: passphrase
    };

    return server = https.createServer(config, restApp)
      .listen(port, function () {
        wsServer.listen(server);
        console.log('Secure WoT server started on port %s', port);
    })
  } else {
    var http = require('http');
    return server = http.createServer(restApp)
```

Start the internal hardware plugins.

If in secure mode, import the HTTPS module.

The actual certificate file of the server

The private key of the server generated earlier

The password of the private key

By passing it the server you create, the WebSocket library will automatically detect and enable TLS support.

Create an HTTPS server using the config object.

```
      .listen(process.env.PORT | port, function () {
        wsServer.listen(server);
        console.log('Unsecure WoT server started on port %s', port);
    })
  }
};

function initPlugins() {
  var LedsPlugin = require('./plugins/internal/ledsPlugin').LedsPlugin;
  var PirPlugin = require('./plugins/internal/pirPlugin').PirPlugin;
  var Dht22Plugin = require('./plugins/internal/dht22Plugin').Dht22Plugin;

  pirPlugin = new PirPlugin({'simulate': true, 'frequency': 5000});
  pirPlugin.start();

  ledsPlugin = new LedsPlugin({'simulate': true, 'frequency': 5000});
  ledsPlugin.start();

  dht22Plugin = new Dht22Plugin({'simulate': true, 'frequency': 5000});
  dht22Plugin.start();
}

module.exports = createServer;

process.on('SIGINT', function () {
  ledsPlugin.stop();
  pirPlugin.stop();
  dht22Plugin.stop();
  console.log('Bye, bye!');
  process.exit();
});
```

Finally, modify the wot.js file to require wot-server-secure.js, and start the server by running `nodewot.js`. Now, go to https://localhost:8484/properties/pir in your browser. You should get a warning saying that the connection is not private. What this really means appears in the small print: `ERR_CERT_AUTHORITY_INVALID`. This means that the certificate was generated by you and not by a certificate authority (CA) trusted by your browser. There are two ways to fix this: you can buy a certificate from a trusted CA, as explained in the next section, or you can tell your computer to trust the certificate you just created. The best way to do this is by adding the certificate to the trust store of your browser. The operation will differ depending on which environment you're using, but here's how to add it to Firefox: click I Understand The Risk (because now you do, don't you?), Add Exception, and finally Confirm Security Exception. Other browsers like Chrome use the trust store of the underlying operating system. Hence, to ensure Chrome accepts your certificate, go to Preferences > Settings > Show Advanced Settings; in HTTPS/SSL click Manage Certificates. This should open the trust store of your operating system, where you can import the certificate. Adding self-signed SSL certificates directly to your operating system[1] will make it much easier for you to develop secure applications for your Pi.

[1] http://blog.getpostman.com/2014/01/28/using-self-signed-certificates-with-postman/

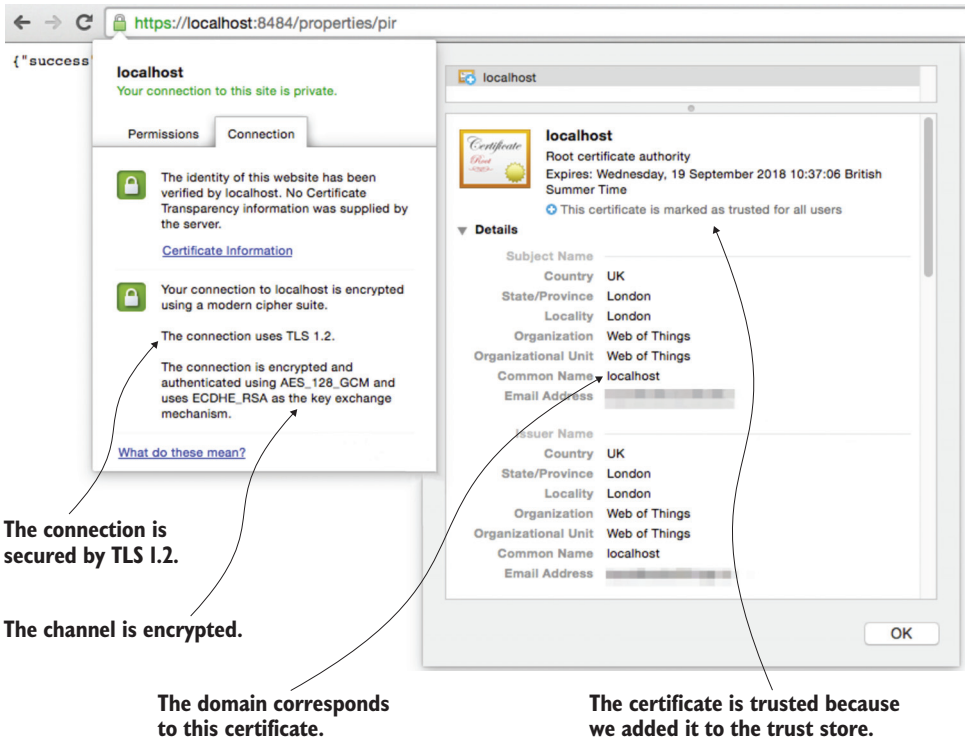


Figure 9.5 The server of the WoT Pi can now be accessed via HTTPS. The details of the secure connection and certificates can be reviewed by clicking the small lock icon on the address bar.

Once your browser trusts the certificate of your WoT Pi, you should be able to get the content returned and the browser should display the usual lock icon on the address bar. If you click it, you'll see the details of your TLS certificate, as shown in figure 9.5.

BEYOND SELF-SIGNED CERTIFICATES

Clearly, having to deal with all these security exceptions isn't nice, but these exceptions exist for a reason: to warn clients that part of the security usually covered by SSL/TLS can't be guaranteed with the certificate you generated. Basically, although the encryption of messages will work with a self-signed certificate (the one you created with the previous command), the authenticity of the server (the Pi) can't be guaranteed. In consequence, the chain of trust is broken—problem 2 of figure 9.2. In an IoT context, this means that attackers could pretend to be the Thing you think you're talking to. This isn't a big deal when your Things are accessible only on the local network, but as soon as you make them available on the web, this becomes critical.

The common way to generate certificates that guarantee the authenticity of the server is to get them from a well-known and trusted certificate authority (CA). There are a number of them on the web, such as Thawte, Symantec, and GeoTrust. The good

thing about certificates issued by such CAs is that they verify who created the certificates, albeit with various degrees of rigor. This means that a client has a greater certainty of which server it's talking to (authentication). In consequence, these certificates, or certificates generated using these, are trusted by a number of clients such as web browsers. More concretely, this means that web browsers and operating systems have these certificates in their trust store.

The problem is that certificates issued by well-known CAs are certainly not free. The business of selling web security is a lucrative one! A direct and unfortunate consequence of this is that a number of sites use cheaper CAs that do a poorer job of checking to whom they deliver certificates, or they decide to not use secured connections at all. But this is rapidly changing: a number of major actors on the web, such as Mozilla, Akamai, Cisco, and the Internet Security Research Group, got together to create the Let's Encrypt[1] project, an automated CA providing free and secure certificates for the public's benefit. There are even ways to automatically generate certificates using Let's Encrypt from a Raspberry Pi running a Node server with Express.[2] Now that you have the basics of TLS, you should consider this when moving your Pi to the World ~~Wild~~ Wide Web.

The nerd corner—I want my Pi to be on the web!

Once the development and testing phase of your WoT Pi is finished, you'll likely want to make it accessible over the web with its own public domain; for example, mypi.webofthings.io. To do this, you could use Yaler,[a] which is a great service and open source project that offers a relay to securely access your embedded devices through your firewall and supports mobile Things connecting to different networks. Alternatively, if you want to go the DIY route, you can use a dynamic DNS service—unless you already have a fixed IP address—that keeps monitoring the IP address of your home router to determine when it changes. There are a number of those, but Duck DNS is straightforward and free. Moreover, it provides clear explanations of how to install it on a Pi.[b] Once this is set up, you'll also have to set up port forwarding on your home router.[c] Then, you might also need to generate (or buy) a certificate with a common name corresponding to the new Duck DNS subdomain of your Pi; for example, mypi.duckdns.org. Once you've done all of this, your Pi should be truly on the world-wide Web of Things. But your Pi will also be ready for attackers to try to hack it, so make sure you protect it well, at the very least by reading to the end of this chapter and implementing the concepts we describe!

[a] https://www.yaler.net/raspberrypi
[b] http://www.duckdns.org/install.jsp#pi
[c] http://portforward.com/

[1] https://letsencrypt.org
[2] https://github.com/DylanPiercey/auto-sni

9.2 Authentication and access control

Once we encrypt the communication between Things and clients as shown in the previous section, we want to enable only some applications to access it. Let's get back to our hotel scenario to understand this issue. The hotel control center application needs to have full access to all devices in the network and the ability to configure and administer them. But Lena, who stays in room 212, only needs to access the devices and services in that room. Besides, she shouldn't be able to configure them, only to send a limited set of commands. First, this means that the Things—or a gateway to which Things are connected—need to be able to know the sender of each request (*identification*). Second, devices need to trust that the sender really is who they claim to be (*authentication*). Third, the devices also need to know if they should accept or reject each request depending on the identity of this sender and which request has been sent (*authorization*). If encryption is like sending a postcard in a sealed envelope, authentication and authorization are like sending that envelope via registered mail: the postman will deliver the letter only to the correct recipient as long as they can prove their identity with a valid ID.

9.2.1 Access control with REST and API tokens

Nowadays, we go through this authentication process all the time on the web, namely every time we enter our username and password on a website. When we use our username/password to log into a website, we initiate a secure session with the server that's stored for a limited time in the server application's memory or in a local browser cookie. During that time, we can send other requests to the server without authenticating again. This method (called *server-based authentication*) is usually stateful because the state of the client is stored on the server. But as you saw in chapter 6, HTTP is a stateless protocol; therefore, using a server-based authentication method goes against this principle and poses certain problems. First, the performance and scalability of the overall systems are limited because each session must be stored in memory and overhead increases when there are many authenticated users. Second, this authentication method poses certain security risks—for example, cross-site request forgery.[1]

To circumvent these issues, an alternative method called *token-based authentication* has become popular and is used by most web APIs. The idea is that a secret token—a long string of characters—that's unique for each client can be used to authenticate each request sent by that client. Because this token is added to the headers or query parameters of each HTTP request sent to the server, all interactions remain stateless. Because no session or state needs to be kept on the server(s), applications can be scaled horizontally without having to worry about where the session of each user is stored.

[1] This method exploits the fact that a malicious website can use your browser to send requests on your behalf to another website you're logged into. See https://en.wikipedia.org/wiki/Cross-site_request_forgery.

Obviously, the API token should be generated using a cryptographically secure pseudo-random generator[1] and should be treated like a password: stored in an encrypted manner.

To generate an API token with Node.js, you can use the `crypto.randomBytes()` function.[2] You'll find the function in the /utils/utils.js file shown in the next listing.

Listing 9.3 utils/utils.js: generate a crypto-secure API token

```
exports.generateApiToken = function(length, chars) {
  if (!length) length = 32;
  if (!chars) chars =
     'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
  var randomBytes = crypto.randomBytes(length);
  var result = new Array(length);

  var cursor = 0;
  for (var i = 0; i < length; i++) {
    cursor += randomBytes[i];
    result[i] = chars[cursor % chars.length];
  }

  return result.join('');
};
```

You can call this function by uncommenting the following line in the http.js file:

```
console.info('Here is a crypto-secure API Key: ' + utils.generateApiToken());
```

When you launch the WoT Pi server, you'll see in the terminal a new API token, which you can copy and paste into the value of the `apiToken` key in the resources/auth.json file. This will be the API token you need to send any request to your Pi.

You'll now modify the WoT Pi application so that for each request that comes in, you check if the request is signed using a valid API token; see the following listing. The best way to do this is to use the middleware pattern shown in the previous section. You'll create an auth.js file in the middleware folder, which has a function that will be called each time a new request comes to your API and which checks if it is signed and valid.

Listing 9.4 auth.js: authentication middleware

```
var keys = require('../resources/auth');

module.exports = function() {
  return function (req, res, next) {
    console.log(req.method + " " + req.path);
    if (req.path.substring(0, 5) === "/css/") {
      next();
```

Allow unauthorized access to the css folder.

[1] https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator
[2] https://nodejs.org/api/crypto.html#crypto_crypto_randombytes_size_callback

```
    } else {
      var token = req.body.token || req.get('authorization') ||
        req.query.token;
      console.log(req.params);
      if (!token) {
        return res.status(401).send({success: false, message: 'API token
          missing.'});
      } else {
        if (token !== keys.apiToken) {
          return res.status(403).send({success: false, message: 'API
            token
            invalid.'});
        } else {
          next();
        }
      }
    }
  }
};
```

Check header or URL parameters or POST body for token.

If no token provided, return 401 UNAUTHORIZED.

If token is not the valid API token, return 403 FORBIDDEN.

If everything is good, save to request for use in other routes.

Finally, you need to add this middleware function to the middleware chain in servers/http.js. Start by requiring the middleware with `auth = require('./../middleware/auth')`, then add it to the chain using `app.use(auth());` right after the CORS middleware. Now, run the WoT server once again and then try accessing https://localhost:8484/properties/pir. You should now get an error message. Try again with https://localhost:8484/properties/pir?token=YOUR_TOKEN (or with Postman by adding the `Authorization` header with your token as value) and it should work: your API now requires a valid token!

In this minimal example, you manually check each request against a hard-coded API token. We wanted to show you the basics of how token-based authentication works, so this is not a robust and scalable solution ready for production applications. You'll need to use a more elaborate solution that suits your use case and devices. Will you have many different users that all need to have their own API token, or is it fine to have only a single token? How granular does your access control need to be? How often will you need to add, remove, or change these permissions? As an exercise, you're welcome to extend this simple token-based implementation to support many users and tokens and also the various end points of your Thing (including WebSockets interactions; see /servers/websocket.js for a solution).

The nerd corner—I want better tokens!

Generating tokens manually and implementing a minimal token-based authentication system from scratch as shown before is a great exercise to help you understand how it works. But for anything more than that, you'll be better off using an actual standard. JSON Web Tokens[a] (JWT) is particularly interesting here because it not only generates

[a] https://jwt.io

secure tokens but also offers a standard mechanism to send encrypted payloads over insecure connections. In other words, JWT makes it possible to send secure content over HTTP and WebSocket packets without using TLS. This is particularly appealing for the WoT because it removes the self-generated certificate warnings in the browser you encountered earlier because certificates aren't required for interactions between an app and Thing within a local network. It's certainly not as standard and battle-tested as TLS, but we've had some promising results in our own tests. There are JWT libraries for many languages including Node.js, so go ahead and give it a try!

9.2.2 OAuth: a web authorization framework

In the previous section, we gave a brief introduction to API tokens, how they work, and how you can implement them on web Things. API tokens are a good starting point, and along with encryption (TLS), they are arguably the bare minimum a WoT device should offer in terms of security. But as soon as we need to share the resources of a device with several users having different authorization rights, simple API tokens like the ones we've introduced present two challenges.

First, we need a process for web applications to generate and retrieve tokens dynamically, ideally through an API. Obviously, we can't just create an API endpoint that returns tokens. This would be insecure and we'd be back where we started because we'd need to secure that API as well. Besides, creating a bespoke mechanism to get tokens wouldn't foster interoperability; it would make the process complicated and bespoke for each device and/or API.

Second, API tokens shouldn't be valid forever. API tokens, just like passwords, should change regularly. We should also be able to invalidate any token manually when needed. This ensures that when an API token has leaked, we can disable it. But again, creating a custom API to renew tokens wouldn't foster interoperability between web clients and web Things.

What to do? It turns out there's a web standard coming to our rescue: OAuth.[1] OAuth is an open standard for authorization and is essentially a mechanism for a web or mobile app to delegate the authentication of a user to a third-party trusted service; for example, Facebook, LinkedIn, or Google. OAuth makes this delegated authentication process secure and simple by dynamically generating access tokens using only web protocols. OAuth also allows sharing resources and between applications. For instance, you can allow some of your Facebook friends to securely access some of your documents on Google.

In short, OAuth standardizes how to authenticate users, generate tokens with an expiration date, regenerate tokens, and provide access to resources in a secure and standard manner over the web. Sound like exactly what we need, doesn't it? Let's see

[1] https://tools.ietf.org/html/rfc6749

how to do this in practice using the most recent version of the OAuth standard: OAuth 2.0.

OAUTH ROLES

A typical OAuth scenario involves four roles:

- *A resource owner*—This is the user who wants to authorize an application to access one of their trusted accounts; for example, your Facebook account.
- *The resource server*—Is the server providing access to the resources the user wants to share? In essence, this is a web API accepting OAuth tokens as credentials.
- *The authorization server*—This is the OAuth server managing authorizations to access the resources. It's a web server offering an OAuth API to authenticate and authorize users. In some cases, the resource server and the authorization server can be the same, such as in the case of Facebook.
- *The application*—This is the web or mobile application that wants to access the resources of the user. To keep the trust chain, the application has to be known by the authorization server in advance and has to authenticate itself using a secret token, which is an API key known only by the authorization server and the application.

The flow of a typical OAuth-delegated authentication mechanism is shown in figure 9.6. At the end of the token exchange process, the application will know who the user is and will be able to access resources on the resource server on behalf of the user. The application can then also renew the token before it expires using an optional refresh token or by running the authorization process again.

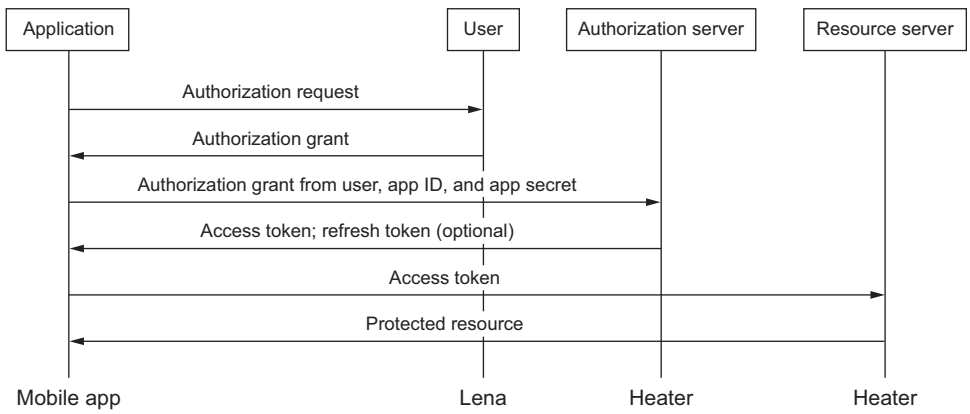


Figure 9.6 OAuth delegated authentication and access flow. The application asks the user if they want to give it access to resources on a third-party trusted service (resource server). If the user accepts, an authorization grant code is generated. This code can be exchanged for an access token with the authorization server. To make sure the authorization server knows the application, the application has to send an app ID and app secret along with the authorization grant code. The access token can then be used to access protected resources within a certain scope from the resource server.

OAuth has become a successful protocol, and as a consequence, a large number of services on the web such as social networks (for example, Facebook, Google+, LinkedIn, and Twitter), developer services (for example, GitHub and BitBucket), and many other websites (such as TripAdvisor and Meetup) support OAuth. But what about the IoT? How does OAuth relate to our web Things?

OAuth and the Web of Things

For a start, if all Things become OAuth servers in place of generating API tokens, web clients will then have a standard way to obtain tokens to access the resources of devices.

Let's get back to our hotel scenario once again. Lena is the user in figure 9.6 and she has a user account on the heater unit of figure 9.2, which is both the authorization server and the resource server. Lena uses a mobile app to control the heater, as shown in figure 9.6. The application asks Lena to log into the heater with her user account, and then the application exchanges the resulting authorization grant for an access token from the heater unit. The heater unit accepts the access token and provides access to the heater to the application on behalf of Lena.

If Lena was interacting with her heater in her home, this would be a practical scenario. But in the case of the hotel, that means that the heater and all other devices would need to know about Lena and all the other hotel clients. Besides, all devices would also need to know all the applications that would interact with them and would need to have generated a secret token for each of them. It's pretty obvious this approach would be a nightmare to maintain!

Implementing an OAuth server on a Linux-based embedded device such as the Pi or the Intel Edison isn't hard because the protocol isn't really heavy. But maintaining the list of all applications, users, and their access scope on each Thing is clearly not going to work and scale for the IoT. We'll look at a better approach in the next section.

The nerd corner—I want my Pi to be an OAuth server!

If you do want to turn your Pi into an OAuth server, go ahead! It will be a good exercise to help you better understand the protocol and will actually make the implementation in the next section more secure. A good place to start is the `node-oauth2-server` Node.js module for Express, which should run seamlessly on your Pi, Edison, or BeagleBone.

9.3 The Social Web of Things

Using OAuth to manage access control to Things is tempting, but not if each Thing has to maintain its own list of users and application. This is where the gateway integration pattern we discovered in chapter 7 can help. What if you had only a single proxy that would know the Things you have at home (or in the entire hotel) and also know the various users involved, so it could manage access control in place of these Things? "But then I still have to create user accounts on this proxy for each user," we hear you

say. Of course, you could do that, but a much better approach would be to use the notion of delegated authentication offered by OAuth, which allows you to use the accounts you already have with OAuth providers you trust, such as Facebook, Twitter, or LinkedIn.

Not only does this approach allow you to reuse the user accounts you already have in other web services, but it also allows you share access to your devices via existing social network relationships. These concepts are often referred to as the Social Web of Things.[1] Let's see what this would look like in more detail. As with all things security, this won't be the easiest ride but will definitely be a rewarding one.

9.3.1 A Social Web of Things authentication proxy

The idea of the Social Web of Things is to create an authentication proxy that controls access to all Things it proxies by identifying users of client applications using trusted third-party services. The detailed steps for this workflow are shown in figure 9.7.

Again, we have four actors: a Thing, a user using a client application, an authentication proxy, and a social network (or any other service with an OAuth server). The client app can use the authentication proxy and the social network to access resources on the Thing. This concept can be implemented in three phases:

1. The first phase is the *Thing proxy trust*. The goal here is to ensure that the proxy can access resources on the Thing securely. If the Thing is protected by an API token (device token), it could be as simple as storing this token on the proxy. If the Thing is also an OAuth server, this step follows an OAuth authentication flow, as shown in figure 9.6. Regardless of the method used to authenticate, after this phase the auth proxy has a secret that lets it access the resources of the Thing.
2. The second phase is the *delegated authentication* step. Here, the user in the client app authenticates via an OAuth authorization server as in figure 9.6. The authentication proxy uses the access token returned by the authorization server to identify the user of the client app and checks to see if the user is authorized to access the Thing. If so, the proxy returns the access token or generates a new one to the client app.
3. The last phase is the *proxied access* step. Once the client app has a token, it can use it to access the resources of the Thing through the authentication proxy. If the token is valid, the authentication proxy will forward the request to the Thing using the secret (device token) it got in phase 1 and send the response back to the client app.

In order not to leak any tokens at any step, all the communication has to be encrypted using TLS. The details for each phase are summarized in figure 9.7.

[1] The Social Web of Things was a concept developed in Dom's thesis (http://webofthings.org/2011/12/01/phd-web-of-things-app-archi/) based on the Friends and Things project: http://webofthings.org/2010/02/02/sharing-in-a-web-of-things/.

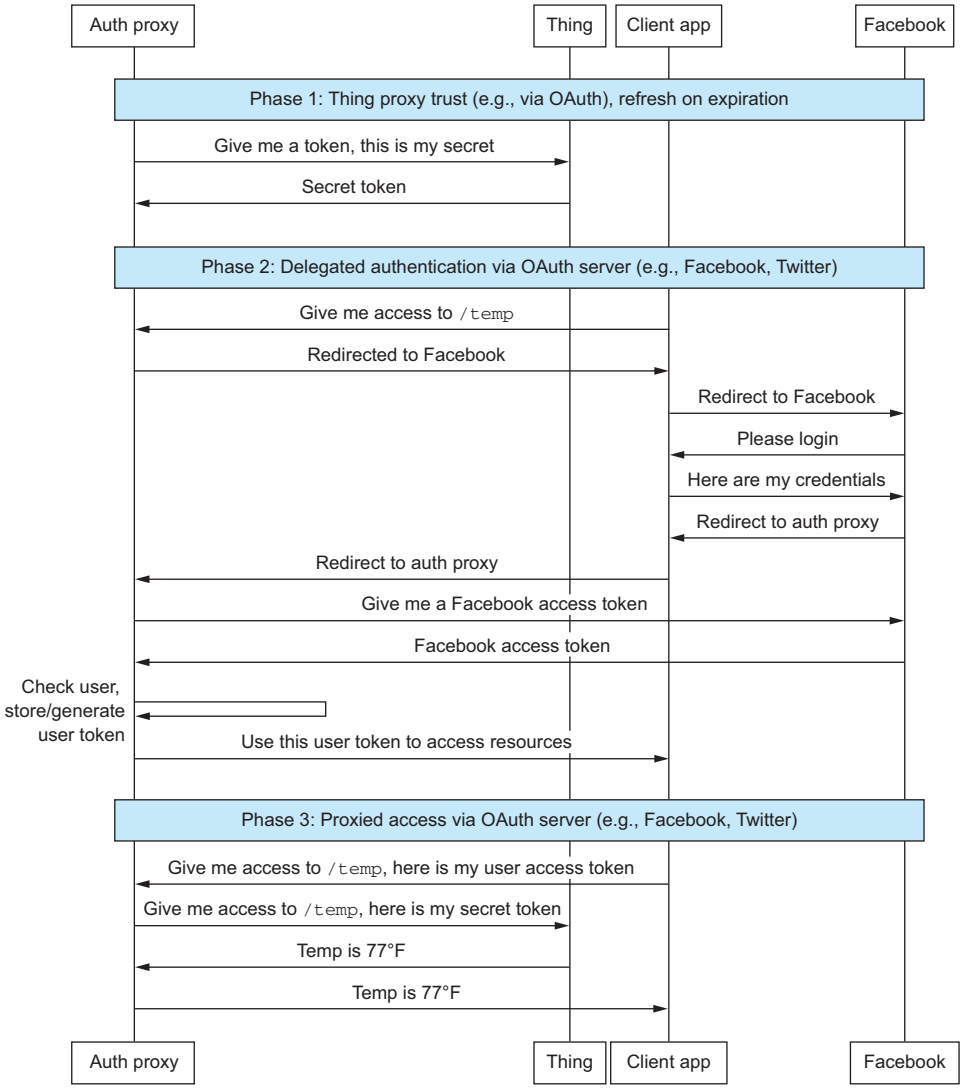


Figure 9.7 Social Web of Things authentication proxy: the auth proxy first establishes a secret with the Thing over a secure channel. Then, a client app requests access to a resource via the auth proxy. It authenticates itself via an OAuth server (here Facebook) and gets back an access token. This token is then used to access resources on the Thing via the auth proxy. For instance, the /temp resource is requested by the client app and given access via the auth proxy forwarding the request to the Thing and relaying the response to the client app.

LEVERAGING SOCIAL NETWORKS

You might have noticed that we overlooked one step in the process: how does the auth proxy know what resources a user can access, or even if they can access any resources at all? Someone needs to configure the proxy with a number of user identifiers

corresponding to the users who can access the system along with a list of resources they can access. In the case of our hotel, we could ask the guests to log in with their Facebook accounts or, even better, with the Booking.com profile they used to book the hotel room in the first place! Then we could save their social identifiers in the auth proxy, along with the paths to the devices in their room. In the case of a home automation system, you can even imagine granting access to lists of friends or relatives. Figure 9.8 is an example of a user interface on the auth proxy that can let you share resources with your friends.

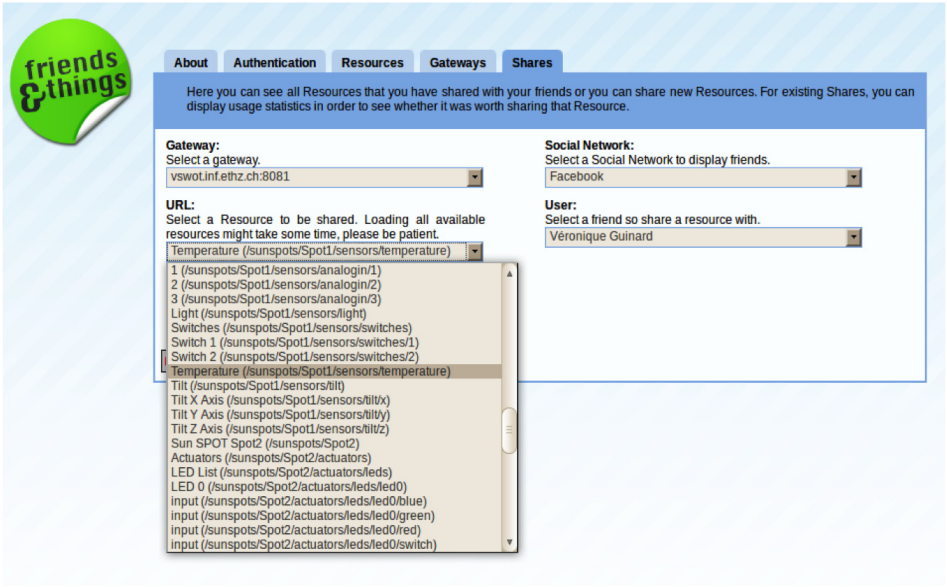


Figure 9.8 User interface of a Social Web of Things authorization proxy: First (upper left), the UI allows the user to select a Thing to be shared and (lower left) the resource of the Thing that should be shared; for example, /temperature. Then (upper right) it lets the owner of the Thing log into their social network, such as Facebook, and (lower right) select a friend to share with or a list of friends. Here we share the temperature sensor of the Spot1 device with Dom's sister via Facebook. [Source: Friends and Things Social Web of Things project[1]]

The good news is that nothing here needs to be hard-coded. Thanks to the fact that our Things speak web (see chapters 5 and 6), we can discover their resources (see chapter 7) and map them to our connections on various OAuth-compliant social networks! This is the very idea of the Social Web of Things: instead of creating abstract access control lists, we can reuse existing social structures as a basis for sharing our Things. Because social networks increasingly reflect our social relationships, we can

[1] hhttp://webofthings.org/2010/02/02/sharing-in-a-web-of-things/

reuse that knowledge to share access to our Things with friends via Facebook, or work colleagues via LinkedIn.

9.3.2 Implementing a Social WoT authentication proxy

Now that you've seen the theory, let's put this into practice and implement a simple authentication proxy for the Social Web of Things, as shown in figure 9.9.

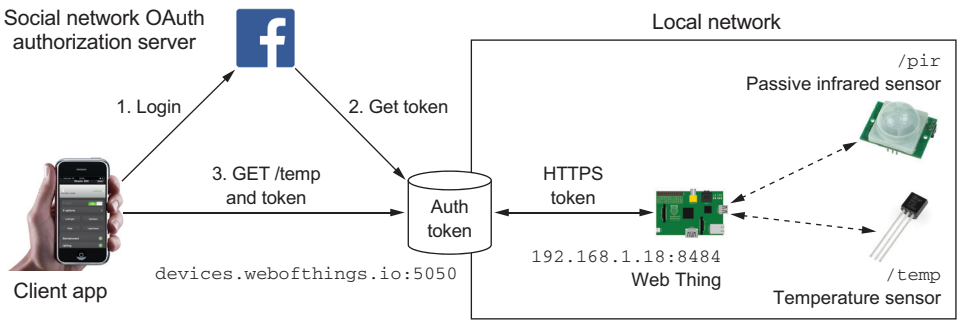


Figure 9.9 A Social Web of Things authentication proxy for your Pi: client apps obtain a token via OAuth on Facebook; this token can then be used to access the Pi resources via the auth proxy. The auth proxy must be accessible on the web, or at least on the same network as the client app, but the Pi can be on a local network as long as the auth proxy can access it.

The complete code for this part is located in the chapter9-sharing/social-auth folder, but we'll only look at the details of some parts here. The proxy could be built directly on top of the WoT Pi code we built in the previous chapters, but as we said before, it makes more sense to implement it as a standalone proxy that can be deployed either on the Pi or somewhere else because it might proxy the access to more than one device.

Creating a Facebook application

Before we can begin coding, we need to make sure that Facebook knows our auth proxy as an authorized Facebook application. To create a Facebook app, you'll need a Facebook account and to apply for a Facebook developer account. If you're not into cat videos or holiday selfies and therefore don't have a Facebook account, feel free to pick another OAuth provider such as Google, Twitter, or GitHub and replace "Facebook" with the OAuth provider you picked in all the following sections. We won't detail how to implement support for other providers, but the principle will be similar, so you shouldn't have too much trouble doing this exercise.

Go to https://developers.facebook.com and apply for a Facebook developer account if you don't already have one. Under My Apps select Register As A Developer. Then you can select My Apps > Add A New App. Select Website, give your app a name,

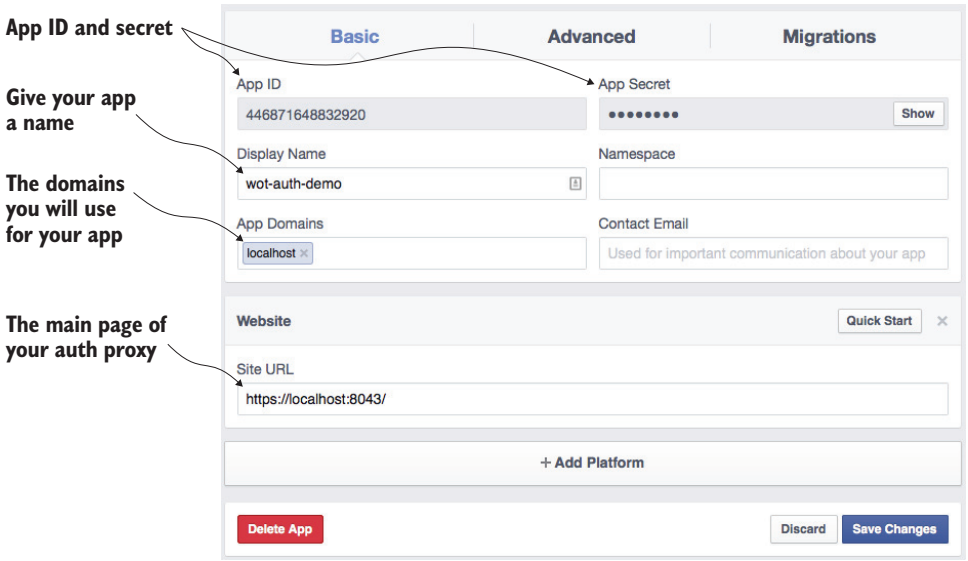


Figure 9.10 Setting up a new Facebook application for our Social WoT auth proxy. The app ID and secret will be used by Facebook to authenticate our app.

and select Skip Quick Start. You should now have a new Facebook app; fill the fields as shown in figure 9.10 by clicking Settings under your app name.

Once you've done this, you can note the two bits of information you need: the app ID and the app secret of your Facebook application. You'll need to send these to Facebook to authenticate your client app. Note that until your app is published publicly, only you and the people you invite as developers/admins will be able to log in via this Facebook app.

PASSPORT.JS: THE AUTHENTICATION MIDDLEWARE FOR EXPRESS

Now that your Facebook app is ready, you need to integrate it into your code. You begin by creating a simple Express application with an HTTPS server. You can find this application in chapter9-sharing/social-auth/authProxy.js, but we won't detail it here because it's similar to the Express apps you created in chapter 8 and the previous sections. Next, you'll create the component that authenticates users via Facebook. You implement it using one of the most popular Node.js modules, Passport.js.[1] Passport is an impressive authentication middleware that provides simple integration of a number of authentication techniques—more than 300!—including OAuth and, hence, all the social networks implementing it.

After installing Passport via `npm install --s passport`, you install the Facebook authentication module of Passport, called a *strategy*, via `npm install --s passport-facebook`. If you want to authenticate via Twitter, LinkedIn, or GitHub, you'll need to

[1] http://passportjs.org/

install the corresponding Passport strategy; for example, `passport-twitter` or `passport-linkedin`. As long as you pick a network that supports OAuth, the implementation of the proxy with your chosen authentication strategy will be almost the same as the one used for Facebook.

Implementing a Facebook authentication strategy

You're now ready to add Facebook authentication support to your proxy. The providers/facebook.js file shows you how to do that. As shown in the following listing, you have to implement a number of functions to work with a Passport strategy.

Listing 9.5 providers/facebook.js: a Facebook authentication strategy

```
var passport = require('passport'), [...]

var acl = require('../config/acl.json');
var facebookAppId = 'YOUR_APP_ID';
var facebookAppSecret = 'YOUR_APP_SECRET';
var socialNetworkName = 'facebook';
var callbackResource = '/auth/facebook/callback';
var callbackUrl = 'https://localhost:' + acl.config.sourcePort +
  callbackResource;

module.exports.setupFacebookAuth = setupFacebookAuth;
function setupFacebookAuth(app) {
  app.use(cookieParser());
  app.use(methodOverride());
  app.use(session({secret: 'keyboard cat',
        resave: true, saveUninitialized: true}));
  app.use(passport.initialize());
  app.use(passport.session());

  passport.serializeUser(function (user, done) {
    done(null, user);
  });

  passport.deserializeUser(function (obj, done) {
    done(null, obj);
  });

  passport.use(new FacebookStrategy({
        clientID: facebookAppId,
        clientSecret: facebookAppSecret,
        callbackURL: callbackUrl
      },
      function (accessToken, refreshToken, profile, done) {

        auth.checkUser(socialId(profile.id), accessToken,
           function (err, res) {
           if (err) return done(err, null);
           else return done(null, profile);
         });
      }));
```

Configuration variables: FB app ID, app secret, name, and the URL to call back after a user authentication on Facebook

Initialize Passport and support storing the user login in sessions.

If you had a database of users, you'd use these two methods to load and save users.

The credentials used to authenticate your auth proxy as a Facebook app

This URL will be called by Facebook after a successful login.

The "verify" function, called by the framework after a successful authentication with the provider; here you check if the user is known by the proxy and store their token if so.

```
  app.get('/auth/facebook',
    passport.authenticate('facebook'),
    function (req, res) {});

  app.get(callbackResource,
    passport.authenticate('facebook', {session: true,
          failureRedirect: '/login'}),
    function (req, res) {
      res.redirect('/account');
    });

  app.get('/account', ensureAuthenticated, function (req, res) {
    auth.getToken(socialId(req.user.id), function (err, user) {
      if (err) res.redirect('/login');
      else {
        req.user.token = user.token;
        res.render('account', {user: req.user});
      }
    });
  });

  function socialId(userId) {
    return socialNetworkName + ':' + userId;
  };
  [...]
};
```

Trigger the authentication process, and redirect the user to facebook.com.

Facebook.com will redirect the user to the callbackUrl, so this function will never be called!

This route will be called by Facebook after user authentication. If it fails you, redirect to /login; otherwise to /account.

If the user is authenticated, you get their token and display their account page; otherwise redirect to /login.

A unique social identifier is formed by concatenating the social userId and the social network name.

At first sight, this flow might seem a bit complex. It consists of a number of routes that redirect the user to a Facebook login page and back from Facebook to your proxy alongside a code that can be exchanged for a token. Passport takes care of all the nitty-gritty details for you. The good news is that all authentication strategies have to implement the same methods, so what you learned here can be applied to other social networks as well!

This was the core of the Facebook authentication mechanism, and now you also need to make sure users have a user interface (HTML views) for all the routes you created. You can certainly write HTML pages from scratch, but it's easier to reuse Handlebars, the templating engine we used in the previous chapters. The pages we created are located in the /views folder. At a minimum you'll need a login.html page with a link to /auth/facebook to trigger the authentication process. You'll also need an account.html page to which the user will be redirected upon a successful Facebook authentication.

IMPLEMENTING ACCESS CONTROL LISTS

Now that your application allows users to authenticate via Facebook using OAuth, you need to decide which user can access which resource on which Thing. In essence, you need to create an access control list (ACL). There are various ways to implement ACLs, such as by storing them in the local database. To keep things simple, you'll use a JSON configuration file, which can be found in config/acl.json and is shown in the next listing. This file keeps track of which users can access which resources on your Pi.

Listing 9.6 config/acl.json: the access control list JSON file

```
{
  "protected": [
    {
      "uid": "facebook:10207489314897153",
      "resources": [
        "/properties", "/properties/temperature",
        "/properties/humidity",
        "/properties/pir", "/leds/1", "/leds/2", "/actions/ledState"
      ]
    }, {...}
  ],
  "open": [
    "/", "/model", "/account", "/login", "/logout", "/auth/facebook",
    "/auth/facebook/callback"
  ],
  "things": [
    {
      "id": "WoTPi",
      "url": "https://127.0.0.1:8484",
      "token": "cKXRTaRylYWQiF3MICaKndG4WJMcVLFz"
    }, {...}
  ],
  "config": {
    "sourcePort" : 5050
  }
}
```

An array of the resources you want to protect

User IDs are concatenations of the social network name and the social network ID.

The list of resources user facebook:10207489314897153 is allowed to access; replace the number with your Facebook ID.

The resources you want to allow access without authentication

The list of Things this proxy covers alongside their root URL and secret token; could also be generated dynamically via OAuth if the Thing supports it.

A difficulty might be finding the user IDs of users you want to share with using their social network identifier. A good way is to ask them to log in first because this will display their social network ID on the account page you got back from Facebook. Alternatively, you can use the Facebook Graph API explorer[1] tool. Make sure you add your own ID in the ACL!

Now that your ACL is in place, you need to check what you get back from Facebook against it to ensure the users who are trying to log in are really welcome. Similarly, you need to check that they can access the Things' resources requested. You implement this using a middleware in /middleware/auth.js, as shown in the next listing.

Listing 9.7 Authorizing user requests: /middleware/auth.js

```
var acl = require('../config/acl.json'), [...]

exports.socialTokenAuth = function (req, res, next) {
  if (isOpen(req.path)) {
    next();
  } else {
    var token = req.body.token || req.param('token') ||
      req.headers['Authorization'];
    if (!token) {
```

Require your ACL config file.

If the request is for an open path, call the next middleware.

[1] https://developers.facebook.com/tools/explorer/

```
      return res.status(401).send({success: false, message: 'API token
        missing.'});
    } else {
      checkUserAcl(token, req.path, function (err, user) {
        if (err) {
          return res.status(403).send({success: false, message: err});
        }
        next();
      });
    }
  }
};

function checkUserAcl(token, path, callback) {
  var userAcl = findInAcl(function (current) {
    return current.token === token && current.resources.indexOf(path)
      !== -1;
  });
  if (userAcl) {
    callback(null, userAcl);
  } else {
    callback('Not authorized for this resource!', null);
  }
};
function findInAcl(filter) {
  return acl.protected.filter(filter)[0];
};

function isOpen(path) {
  [...] if (acl.open.indexOf(path) !== -1) return true;
}

exports.checkUser = checkUser;
function checkUser(socialUserId, token, callback) {
  var result = findInAcl(function(current) {
    return current.uid === socialUserId;
  });
  if(result) {
    result.token = token;
    callback(null, result);
  } else {
    callback('User not found!', null);
  }
};
[...]
```

Otherwise, get the access token and check the ACL for this token.

Otherwise, the user is good to go, and you call the next middleware.

**eturn a
ıs code.**

Can we find a user with the given token and the given path?

Handle open resources.

Called by facebook.js when a user is authenticated

If the user ID you got from Facebook is present in your ACL, you have a winner!

Store the user token to allow them to make subsequent calls to resources they can access.

PROXYING RESOURCES OF THINGS

Finally, you need to implement the actual proxying: once a request is deemed valid by the middleware, you need to contact the Thing that serves this resource and proxy the results back to the client. This part is no different from any other HTTP proxy. To implement it, you'll use a blazing-fast Node module for building proxies called

`node-http-proxy`.[1] Install it via `npm install --save http-proxy`. Then use this module to build another middleware in /middleware/proxy.js, as shown in the next listing.

Listing 9.8 Proxying requests to Things: /middleware/proxy.js

```
var https = require('https'),
  fs = require('fs'),
  config = require('../config/acl.json').things[0],
  httpProxy = require('http-proxy');

var proxyServer = httpProxy.createProxyServer({
  ssl: {
    key: fs.readFileSync('./config/change_me_privateKey.pem', 'utf8'),
    cert: fs.readFileSync('./config/change_me_caCert.pem', 'utf8'),
    passphrase: 'webofthings'
  },
  secure: false
});

module.exports = function() {
  return function proxy(req, res, next) {
    req.headers['authorization'] = config.token;
    proxyServer.web(req, res, {target: config.url});
  }
};
```

Load the Thing that can be proxied (there's only one here).

Initialize the proxy server, making it an HTTPS proxy to ensure end-to-end encryption.

Do not verify the certificate (true would refuse a local certificate).

Proxy the request; notice that this middleware doesn't call next() because it should be the last in the chain.

Proxy middleware function; add the secret token of the Thing.

That's it! You should now have a full Social Web of Things authentication proxy. To test it, run `node authProxy.js`. Then, start the WoT Pi using `node wot.js` with simple token authentication enabled, as shown in section 9.2.1, or with OAuth if you implemented it.

Try to access a resource of your Pi via the proxy with an invalid token; for example, https://raspberrypi.local:5050/properties/pir?token=1234. As expected, this will return an error: `Not authorized for this resource!`

Now, let's get an access token to issue a valid request: start by browsing to https://IP:5050/login. This should prompt you to log in on Facebook (if your browser doesn't have a Facebook login cookie sitting in the cupboard) and then should ask you if you authorize the proxy Facebook app to access your profile. If you accept, you'll land on your profile page, as shown in figure 9.11, where you can see your access token. Copy it and open https://raspberrypi.local:5050/properties/pir?token=YOUR-TOKEN once again, but this time with your new token. If everything works, you should get the HTML representation of the PIR sensor. Take a deep breath and think about what you just did: you merged the Social Web with the physical world!

[1] https://github.com/nodejitsu/node-http-proxy

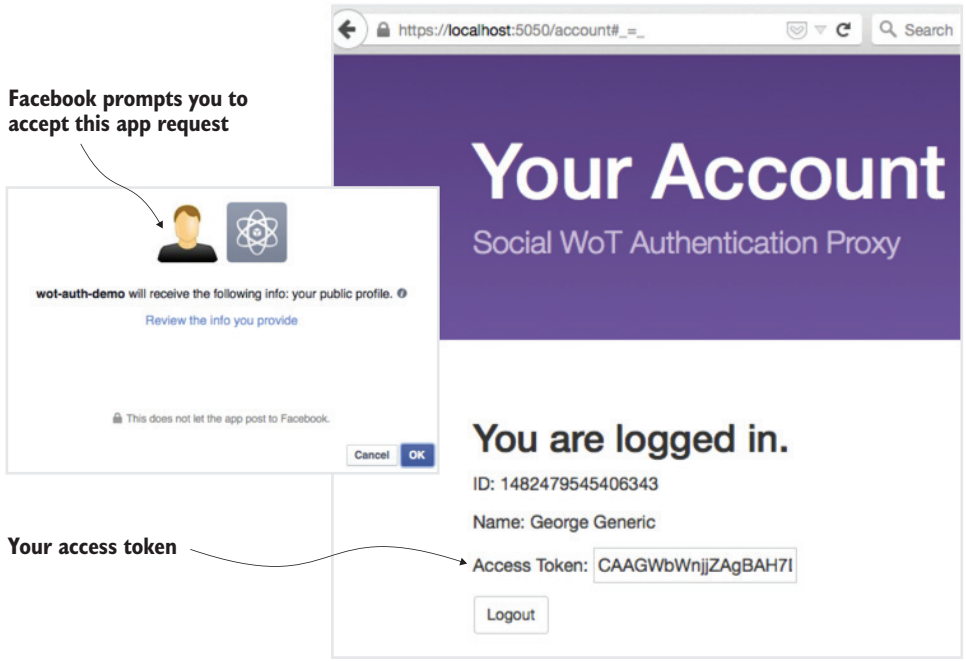


Figure 9.11 First, Facebook will prompt the user to accept the application request. After a successful Facebook authentication, the user is redirected to their Account page on the auth proxy, where they can retrieve their access token to be used in subsequent calls to the Things behind the proxy.

The nerd corner—I want more of this!

As usual, there are many possible ways of extending this example because we kept the implementation simple. Here are a few extension ideas: you could use what you learned in chapter 8 to implement a system for the proxy to automatically discover web Thing model-compliant Things. You could also make the ACL much easier to deal with by implementing wildcards; for example, `/properties/*`. Or you could create a UI for the proxy that lets you share with your friends or that lets you add authorized users dynamically (for our hotel scenario). If you're still hungry for more, you could also implement proxying for WebSockets; `node-http-proxy` supports it as well. Finally, you could implement an OAuth server on the Pi—for example, using `node-oauth2-server`–and change the proxy to dynamically get an OAuth access token from the Pi instead of a simple token; this would make the flow more secure and much more flexible.

9.4 Beyond the book

In this chapter, you learned how to blend the Social Web and the Web of Things to get to the Social Web of Things. Not a small achievement for a single chapter! Although you should certainly enjoy the moment, you should also realize that we barely

scratched the surface of security for the IoT and the WoT. We didn't cover a number of aspects, from ensuring privacy to protecting Things against distributed denial of service attacks or securing software and firmware updates.

By definition, perfect security is unattainable. Securing computer networks is a constant battle between security experts and hackers, where security systems always need to be one step ahead because the better our machines and tools get, the easier it is to break secure systems. Network security should be a constant discipline rather than a one-off event, and you need to keep informed and updated as you pursue your IoT adventure.[1]

As the IoT moves out of its teenage years and into adulthood, different focal points will appear. First, security will become ubiquitous and a must-have, rather than a nice-to-have. But just as HTTP might be too heavy for resource-limited devices, security protocols such as TLS and their underlying cypher suites are too heavy for the most resource-constrained devices. This is why lighter-weight versions of TLS are being developed, such as DTLS,[2] which is similar to TLS but runs on top of UDP instead of TCP and also has a smaller memory footprint. Although such protocols represent interesting evolutions, some researchers are looking at revolutions! For example, some researchers started looking at a concept they refer to as *device democracy*.[3] In this model, devices become more autonomous and favor peer-to-peer interactions over centralized cloud services. Security is ensured using a blockchain mechanism: similar to the way bitcoin transactions are validated by a number of independent parties in the bitcoin network, devices could all participate in making the IoT secure. Without a doubt, IoT security will change drastically in the next few years, as the web itself will evolve to match today's needs.

The nerd corner—I want the future of secure application management!

As mentioned before, managing applications or firmware updates on an embedded device can be tricky to get right and secure: if you don't do it right, such as by using an insecure HTTP server, attackers could use your update mechanism to inject whatever they like on your customers' devices! Luckily, as the IoT matures, interesting, secure, and scalable solutions appear to help you deploy code on your Things. As an example, resin.io lets you use Git to push new versions of your code to all your Things or to a selection of them. It also uses Docker containers to package and run

[1] Some good bedside readings:
- http://h30499.www3.hp.com/t5/Fortify-Application-Security/HP-Study-Reveals-70-Percent-of-Internet-of-Things-Devices/ba-p/6556284
- https://www.owasp.org/index.php/Main_Page
- http://arstechnica.com/security/2016/01/how-to-search-the-internet-of-things-for-photos-of-sleeping-babies/
- http://techcrunch.com/2015/10/24/why-iot-security-is-so-critical

[2] https://en.wikipedia.org/wiki/Datagram_Transport_Layer_Security

[3] http://www-935.ibm.com/services/us/gbs/thoughtleadership/internetofthings/

(continued)

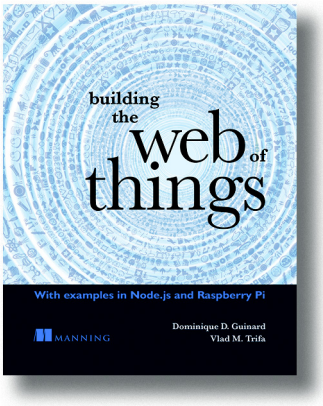
several applications in isolation on embedded devices, which improves portability, security, and stability. Finally, it works well with Node.js and the Pi and is free if you have a small number of devices, so go ahead and try it.[a]

[a] https://resin.io

9.5 Summary

- You must cover four basic principles to secure IoT systems: encrypted communication, server authentication, client authentication, and access control.
- Encrypted communication ensures attackers can't read the content of messages. It uses encryption mechanisms based on symmetric or asymmetric keys.
- You should use TLS to encrypt messages on the web. TLS is based on asymmetric keys: a public key and a private server key.
- Server authentication ensures attackers can't pretend to be the server. On the web, this is achieved by using SSL (TLS) certificates. The delivery of these certificates is controlled through a chain of trust where only trusted parties called certificate authorities can deliver certificates to identify web servers.
- Instead of buying certificates from a trusted third party, you can create self-signed TLS certificates on a Raspberry Pi. The drawback is that web browsers will flag the communication as unsecure because they don't have the CA certificate in their trust store.
- You can achieve client authentication using simple API tokens. Tokens should rotate on a regular basis and should be generated using crypto secure random algorithms so that their sequence can't be guessed.
- The OAuth protocol can be used to generate API tokens in a dynamic, standard, and secure manner and is supported by many embedded Linux devices such as the Raspberry Pi.
- The delegated authentication mechanism of OAuth relies on other OAuth providers to authenticate users and create API tokens. As an example, a user of a Thing can be identified using Facebook via OAuth.
- You can implement access control for Things to reflect your social contacts by creating an authentication proxy using OAuth for clients' authentication and contacts from social networks.

Now that you've seen how to secure your web-connected Things so that their data and services can be securely shared and accessed over the web, it's time to move to the final layer of the WoT architecture: Compose. In the next chapter, you'll see how to take all the components you learned about in this book and combine them to build a whole new generation of web applications: physical mashups. Integrating real-time data from numerous physical sources directly within web applications and services is without doubt the future of the web. We want to make sure you have the tools you need to get there in no time!



Building the Web of Things is a hands-on guide that will teach how to design and implement scalable, flexible, and open IoT solutions using Web technologies. This book focuses on providing the right balance of theory, code samples and practical examples, to enable you how to successfully connect all sorts of devices to the Web and how to expose their services and data over REST APIs. After you build a simple proof of concept app, you'll learn a systematic methodology and system architecture for connecting things to the Web, finding other things, sharing data, and combining these components to rapidly build distributed applications and physical mashups. Each chapter will help you gain the knowledge and skills you'll need to fully take advantage of a new generation of real-time, web-connected devices and services and to be able to build scalable applications that merge the physical and digital worlds.

The Internet of Things (IoT) is a hot conversation topic. Analysts call it a disruptive technology. Competing standards and technologies are appearing daily, and there are no tangible signs of a single protocol that will enable all devices, services, and applications to talk to each other seamlessly. Fortunately, there's a great universal IoT application platform available now: the World Wide Web. Web standards and tools provide the ideal substrate for connected devices and applications to exchange data, and this vision is called the Web of Things.

What's inside

- Using Web technologies to sense and connect the real world
- Rapidly build a Web interface to control your Smart Home using a Raspberry Pi
- Use the real-time, programmable, social and semantic Web to build a Web API for any device
- Build real-time physical mashups with JavaScript and node.js
- Learn how to integrate other protocols such as MQTT, CoAP or Bluetooth to the Web
- Learn how to use a variety WoT and IoT platforms, tools, and protocols

Whether you're a seasoned developer, a system architect, or a curious amateur with basic programming skills, this book will provide you with a complete toolbox to become an active participant in the Web of Things revolution.

What Is Amazon Web Services?

As web applications increase in size and complexity, traditional data centers are moving much of their operations to cloud-based systems like Amazon Web Services. These services provide stable computing resources that replace physical servers, along with numerous services that can simplify application development and operational management concerns such as scaling and security. This chapter gives an overview of Amazon Web Services.

Chapter 1 from *Amazon Web Services in Action* by Michael Wittig and Andreas Wittig

What is Amazon Web Services?

This chapter covers

- Overview of Amazon Web Services
- Benefits of using Amazon Web Services
- Examples of what you can do with Amazon Web Services
- Creating and setting up an Amazon Web Services account

Amazon Web Services (AWS) is a platform of web services offering solutions for computing, storing, and networking, at different layers of abstraction. You can use these services to host web sites, run enterprise applications, and mine tremendous amounts of data. The term *web service* means services can be controlled via a web interface. The web interface can be used by machines or by humans via a graphical user interface. The most prominent services are EC2, which offers virtual servers, and S3, which offers storage capacity. Services on AWS work well together; you can use them to replicate your existing on-premises setup or design a new setup from scratch. Services are charged for on a pay-per-use pricing model.

As an AWS customer, you can choose among different data centers. AWS data centers are distributed in the United States, Europe, Asia, and South America. For example, you can start a virtual server in Japan in the same way you can start a virtual server in Ireland. This enables you to serve customers worldwide with a global infrastructure.

The map in figure 1.1 shows the data centers available to all customers.

Which hardware powers AWS?

AWS keeps secret the hardware used in its data centers. The scale at which AWS operates computing, networking, and storage hardware is tremendous. It probably uses commodity components to save money compared to hardware that charges extra for a brand name. Handling of hardware failure is built into real-world processes and software.[1]

AWS also uses hardware especially developed for its use cases. A good example is the Xeon E5-2666 v3 CPU from Intel. This CPU is optimized to power virtual servers from the c4 family.

In more general terms, AWS is known as a *cloud computing platform*.

1.1 What is cloud computing?

Almost every IT solution is labeled with the term *cloud computing* or just *cloud* nowadays. A buzzword may help to sell, but it's hard to work with in a book.

Cloud computing, or the cloud, is a metaphor for supply and consumption of IT resources. The IT resources in the cloud aren't directly visible to the user; there are layers of abstraction in between. The level of abstraction offered by the cloud may vary from virtual hardware to complex distributed systems. Resources are available on demand in enormous quantities and paid for per use.

Figure 1.1 AWS data center locations

[1] Bernard Golden, "Amazon Web Services (AWS) Hardware," *For Dummies*, http://mng.bz/k6lT.

Here's a more official definition from the National Institute of Standards and Technology:

> *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*
>
> —The NIST Definition of Cloud Computing, National Institute of Standards and Technology

Clouds are often divided into the following types:

- *Public*—A cloud managed by an organization and open to use by the general public
- *Private*—A cloud that virtualizes and shares the IT infrastructure within a single organization
- *Hybrid*—A mixture of a public and a private cloud

AWS is a public cloud. Cloud computing services also have several classifications:

- *Infrastructure as a service (IaaS)*—Offers fundamental resources like computing, storage, and networking capabilities, using virtual servers such as Amazon EC2, Google Compute Engine, and Microsoft Azure virtual machines
- *Platform as a service (PaaS)*—Provides platforms to deploy custom applications to the cloud, such as AWS Elastic Beanstalk, Google App Engine, and Heroku
- *Software as a service (SaaS)*—Combines infrastructure and software running in the cloud, including office applications like Amazon WorkSpaces, Google Apps for Work, and Microsoft Office 365

The AWS product portfolio contains IaaS, PaaS, and SaaS. Let's take a more concrete look at what you can do with AWS.

1.2 What can you do with AWS?

You can run any application on AWS by using one or a combination of services. The examples in this section will give you an idea of what you can do with AWS.

1.2.1 Hosting a web shop

John is CIO of a medium-sized e-commerce business. His goal is to provide his customers with a fast and reliable web shop. He decided to host the web shop on-premises, and three years ago he rented servers in a data center. A web server handles requests from customers, and a database stores product information and orders. John is evaluating how his company can take advantage of AWS by running the same setup on AWS, as shown in figure 1.2.

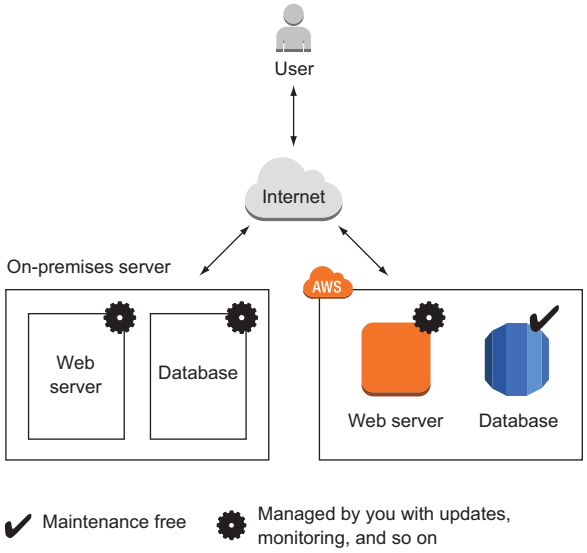


Figure 1.2 Running a web shop on-premises vs. on AWS

John realized that other options are available to improve his setup on AWS with additional services:

- The web shop consists of dynamic content (such as products and their prices) and static content (such as the company logo). By splitting dynamic and static content, John reduced the load for his web servers and improved performance by delivering the static content over a content delivery network (CDN).
- John uses maintenance-free services including a database, an object store, and a DNS system on AWS. This frees him from managing these parts of the system, decreases operational costs, and improves quality.
- The application running the web shop can be installed on virtual servers. John split the capacity of the old on-premises server into multiple smaller virtual servers at no extra cost. If one of these virtual servers fails, the load balancer will send customer requests to the other virtual servers. This setup improves the web shop's reliability.

Figure 1.3 shows how John enhanced the web shop setup with AWS.

John started a proof-of-concept project and found that his web application can be transferred to AWS and that services are available to help improve his setup.

1.2.2 Running a Java EE application in your private network

Maureen is a senior system architect in a global corporation. She wants to move parts of the business applications to AWS when the company's data-center contract expires in a few months, to reduce costs and gain flexibility. She found that it's possible to run enterprise applications on AWS.

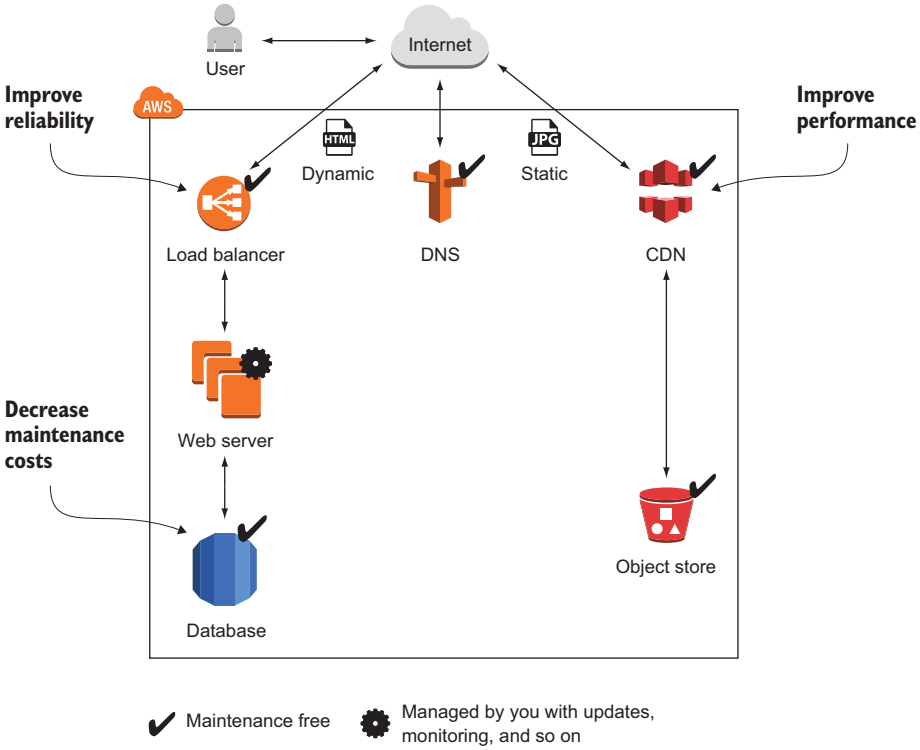


Figure 1.3 Running a web shop on AWS with CDN for better performance, a load balancer for high availability, and a managed database to decrease maintenance costs

To do so, she defines a virtual network in the cloud and connects it to the corporate network through a virtual private network (VPN) connection. The company can control access and protect mission-critical data by using subnets and control traffic between them with access-control lists. Maureen controls traffic to the internet using Network Address Translation (NAT) and firewalls. She installs application servers on virtual machines (VMs) to run the Java EE application. Maureen is also thinking about storing data in a SQL database service (such as Oracle Database Enterprise Edition or Microsoft SQL Server EE). Figure 1.4 illustrates Maureen's architecture.

Maureen has managed to connect the on-premises data center with a private network on AWS. Her team has already started to move the first enterprise application to the cloud.

1.2.3 Meeting legal and business data archival requirements

Greg is responsible for the IT infrastructure of a small law office. His primary goal is to store and archive all data in a reliable and durable way. He operates a file server to offer the possibility of sharing documents within the office. Storing all the data is a challenge for him:

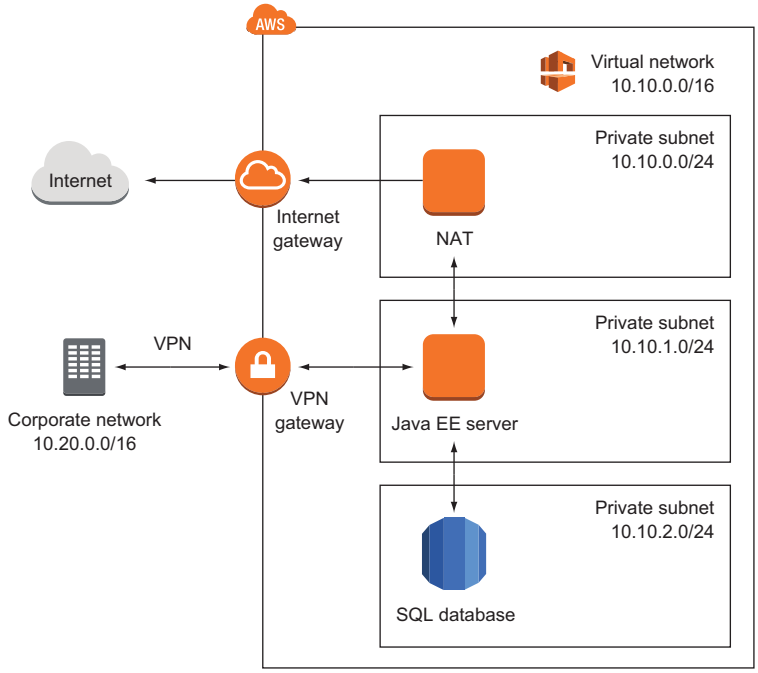


Figure 1.4 Running a Java EE application with enterprise networking on AWS

- He needs to back up all files to prevent the loss of critical data. To do so, Greg copies the data from the file server to another network-attached storage, so he had to buy the hardware for the file server twice. The file server and the backup server are located close together, so he is failing to meet disaster-recovery requirements to recover from a fire or a break-in.
- To meet legal and business data archival requirements, Greg needs to store data for a long time. Storing data for 10 years or longer is tricky. Greg uses an expensive archive solution to do so.

To save money and increase data security, Greg decided to use AWS. He transferred data to a highly available object store. A storage gateway makes it unnecessary to buy and operate network-attached storage and a backup on-premises. A virtual tape deck takes over the task of archiving data for the required length of time. Figure 1.5 shows how Greg implemented this use case on AWS and compares it to the on-premises solution.

Greg is fine with the new solution to store and archive data on AWS because he was able to improve quality and he gained the possibility of scaling storage size.

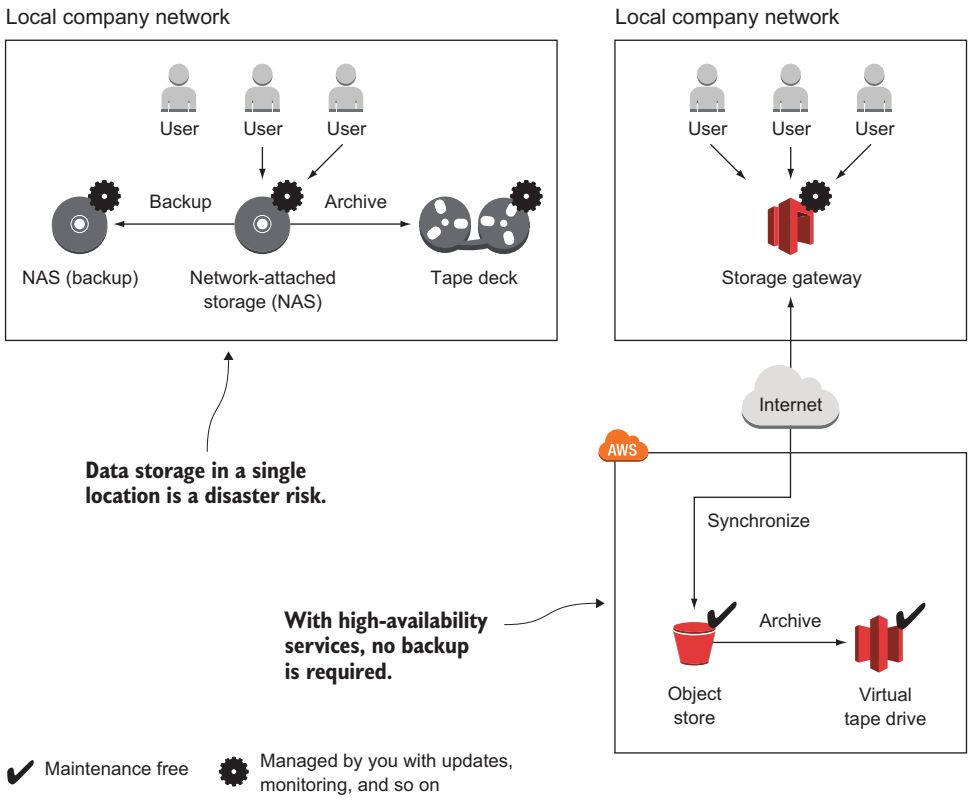


Figure 1.5 Backing up and archiving data on-premises and on AWS

1.2.4 Implementing a fault-tolerant system architecture

Alexa is a software engineer working for a fast-growing startup. She knows that Murphy's Law applies to IT infrastructure: anything that can go wrong, will go wrong. Alexa is working hard to build a fault-tolerant system to prevent outages from ruining the business. She knows that there are two type of services on AWS: fault-tolerant services and services that can be used in a fault-tolerant way. Alexa builds a system like the one shown in figure 1.6 with a fault-tolerant architecture. The database service is offered with replication and failover handling. Alexa uses virtual servers acting as web servers. These virtual servers aren't fault tolerant by default. But Alexa uses a load balancer and can launch multiple servers in different data centers to achieve fault tolerance.

So far, Alexa has protected the startup from major outages. Nevertheless, she and her team are always planning for failure.

You now have a broad idea of what you can do with AWS. Generally speaking, you can host any application on AWS. The next section explains the nine most important benefits AWS has to offer.

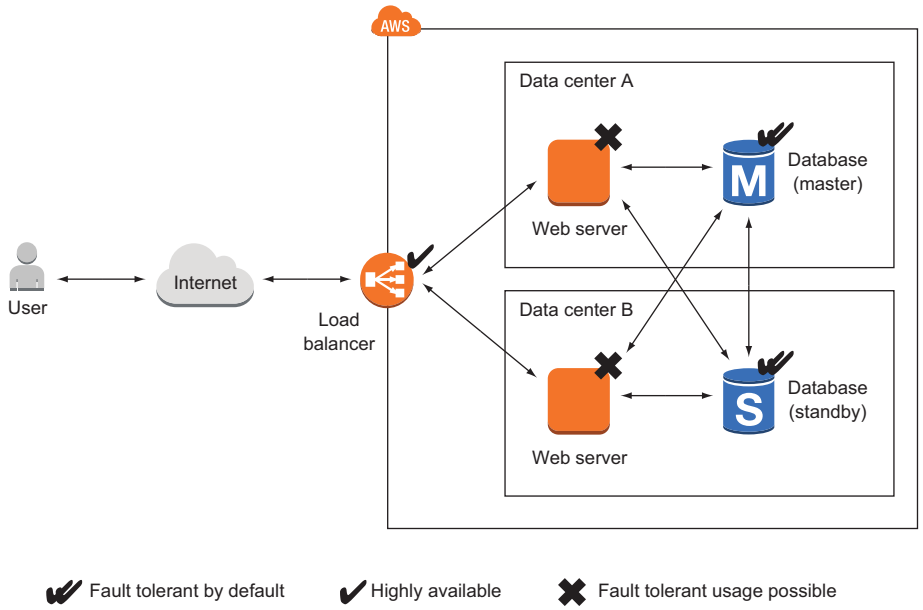


Figure 1.6 Building a fault-tolerant system on AWS

1.3 How you can benefit from using AWS

What's the most important advantage of using AWS? Cost savings, you might say. But saving money isn't the only advantage. Let's look at other ways you can benefit from using AWS.

1.3.1 Innovative and fast-growing platform

In 2014, AWS announced more than 500 new services and features during its yearly conference, re:Invent at Las Vegas. On top of that, new features and improvements are released every week. You can transform these new services and features into innovative solutions for your customers and thus achieve a competitive advantage.

The number of attendees to the re:Invent conference grew from 9,000 in 2013 to 13,500 in 2014.[1] AWS counts more than 1 million businesses and government agencies among its customers, and in its Q1 2014 results discussion, the company said it will continue to hire more talent to grow even further.[2] You can expect even more new features and services in the coming years.

[1] Greg Bensinger, "Amazon Conference Showcases Another Side of the Retailer's Business," *Digits*, Nov. 12, 2014, http://mng.bz/hTBo.

[2] "Amazon.com's Management Discusses Q1 2014 Results - Earnings Call Transcript," *Seeking Alpha*, April 24, 2014, http://mng.bz/60qX.

1.3.2 Services solve common problems

As you've learned, AWS is a platform of services. Common problems such as load balancing, queuing, sending email, and storing files are solved for you by services. You don't need to reinvent the wheel. It's your job to pick the right services to build complex systems. Then you can let AWS manage those services while you focus on your customers.

1.3.3 Enabling automation

Because AWS has an API, you can automate everything: you can write code to create networks, start virtual server clusters, or deploy a relational database. Automation increases reliability and improves efficiency.

The more dependencies your system has, the more complex it gets. A human can quickly lose perspective, whereas a computer can cope with graphs of any size. You should concentrate on tasks a human is good at—describing a system—while the computer figures out how to resolve all those dependencies to create the system. Setting up an environment in the cloud based on your blueprints can be automated with the help of infrastructure as code, covered in chapter 4.

1.3.4 Flexible capacity (scalability)

Flexible capacity frees you from planning. You can scale from one server to thousands of servers. Your storage can grow from gigabytes to petabytes. You no longer need to predict your future capacity needs for the coming months and years.

If you run a web shop, you have seasonal traffic patterns, as shown in figure 1.7. Think about day versus night, and weekday versus weekend or holiday. Wouldn't it be nice if you could add capacity when traffic grows and remove capacity when traffic shrinks? That's exactly what flexible capacity is about. You can start new servers within minutes and throw them away a few hours after that.

The cloud has almost no capacity constraints. You no longer need to think about rack space, switches, and power supplies—you can add as many servers as you like. If your data volume grows, you can always add new storage capacity.

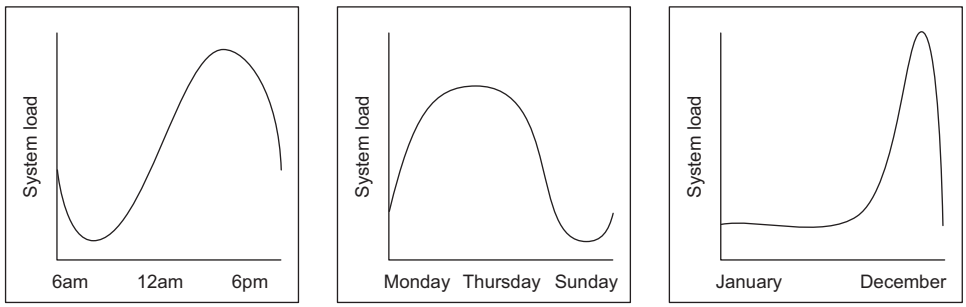


Figure 1.7 Seasonal traffic patterns for a web shop

Flexible capacity also means you can shut down unused systems. In one of our last projects, the test environment only ran from 7:00 a.m. to 8:00 p.m. on weekdays, allowing us to save 60%.

1.3.5 Built for failure (reliability)

Most AWS services are fault-tolerant or highly available. If you use those services, you get reliability for free. AWS supports you as you build systems in a reliable way. It provides everything you need to create your own fault-tolerant systems.

1.3.6 Reducing time to market

In AWS, you request a new virtual server, and a few minutes later that virtual server is booted and ready to use. The same is true with any other AWS service available. You can use them all on demand. This allows you to adapt your infrastructure to new requirements very quickly.

Your development process will be faster because of the shorter feedback loops. You can eliminate constraints such as the number of test environments available; if you need one more test environment, you can create it for a few hours.

1.3.7 Benefiting from economies of scale

At the time of writing, the charges for using AWS have been reduced 42 times since 2008:

- In December 2014, charges for outbound data transfer were lowered by up to 43%.
- In November 2014, charges for using the search service were lowered by 50%.
- In March 2014, charges for using a virtual server were lowered by up to 40%.

As of December 2014, AWS operated 1.4 million servers. All processes related to operations must be optimized to operate at that scale. The bigger AWS gets, the lower the prices will be.

1.3.8 Worldwide

You can deploy your applications as close to your customers as possible. AWS has data centers in the following locations:

- United States (northern Virginia, northern California, Oregon)
- Europe (Germany, Ireland)
- Asia (Japan, Singapore)
- Australia
- South America (Brazil)

With AWS, you can run your business all over the world.

1.3.9 Professional partner

AWS is compliant with the following:

- *ISO 27001*—A worldwide information security standard certified by an independent and accredited certification body

- *FedRAMP & DoD CSM*—Ensures secure cloud computing for the U.S. Federal Government and the U.S. Department of Defense
- *PCI DSS Level 1*—A data security standard (DSS) for the payment card industry (*PCI*) to protect cardholders data
- *ISO 9001*—A standardized quality management approach used worldwide and certified by an independent and accredited certification body

If you're still not convinced that AWS is a professional partner, you should know that Airbnb, Amazon, Intuit, NASA, Nasdaq, Netflix, SoundCloud, and many more are running serious workloads on AWS.

The cost benefit is elaborated in more detail in the next section.

1.4 How much does it cost?

A bill from AWS is similar to an electric bill. Services are billed based on usage. You pay for the hours a virtual server was running, the used storage from the object store (in gigabytes), or the number of running load balancers. Services are invoiced on a monthly basis. The pricing for each service is publicly available; if you want to calculate the monthly cost of a planned setup, you can use the AWS Simple Monthly Calculator (http://aws.amazon.com/calculator).

1.4.1 Free Tier

You can use some AWS services for free during the first 12 months after you sign up. The idea behind the Free Tier is to enable you to experiment with AWS and get some experience. Here is what's included in the Free Tier:

- 750 hours (roughly a month) of a small virtual server running Linux or Windows. This means you can run one virtual server the whole month or you can run 750 virtual servers for one hour.
- 750 hours (or roughly a month) of a load balancer.
- Object store with 5 GB of storage.
- Small database with 20 GB of storage, including backup.

If you exceed the limits of the Free Tier, you start paying for the resources you consume without further notice. You'll receive a bill at the end of the month. We'll show you how to monitor your costs before you begin using AWS. If your Free Tier ends after one year, you pay for all resources you use.

You get some additional benefits, as detailed at http://aws.amazon.com/free. This book will use the Free Tier as much as possible and will clearly state when additional resources are required that aren't covered by the Free Tier.

1.4.2 Billing example

As mentioned earlier, you can be billed in several ways:

- *Based on hours of usage*—If you use a server for 61 minutes, that's usually counted as 2 hours.

- *Based on traffic*—Traffic can be measured in gigabytes or in number of requests.
- *Based on storage usage*—Usage can be either provisioned capacity (for example, 50 GB volume no matter how much you use) or real usage (such as 2.3 GB used).

Remember the web shop example from section 1.2? Figure 1.8 shows the web shop and adds information about how each part is billed.

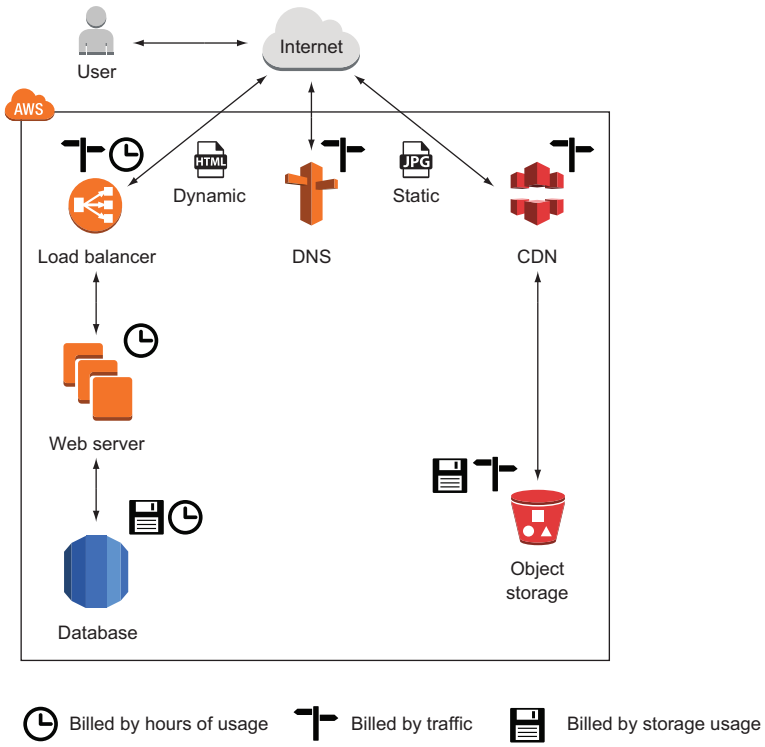


Figure 1.8 Web shop billing example

Let's assume your web shop started successfully in January, and you decided to run a marketing campaign to increase sales for the next month. Lucky you: you were able to increase the number of visitors of your web shop fivefold in February. As you already know, you have to pay for AWS based on usage. Table 1.1 shows your bills for January and February. The number of visitors increased from 100,000 to 500,000, and your monthly bill increased from 142.37 USD to 538.09 USD, which is a 3.7-fold increase. Because your web shop had to handle more traffic, you had to pay more for services, such as the CDN, the web servers, and the database. Other services, like the storage of static files, didn't experience more usage, so the price stayed the same.

Table 1.1 How an AWS bill changes if the number of web shop visitors increases

Service	January usage	February usage	February charge	Increase
Visits to website	100,000	500,000		
CDN	26 M requests + 25 GB traffic	131 M requests + 125 GB traffic	113.31 USD	90.64 USD
Static files	50 GB used storage	50 GB used storage	1.50 USD	0.00 USD
Load balancer	748 hours + 50 GB traffic	748 hours + 250 GB traffic	20.30 USD	1.60 USD
Web servers	1 server = 748 hours	4 servers = 2,992 hours	204.96 USD	153.72 USD
Database (748 hours)	Small server + 20 GB storage	Large server + 20 GB storage	170.66 USD	128.10 USD
Traffic (outgoing traffic to internet)	51 GB	255 GB	22.86 USD	18.46 USD
DNS	2 M requests	10 M requests	4.50 USD	3.20 USD
Total cost			538.09 USD	395.72 USD

With AWS, you can achieve a linear relationship between traffic and costs. And other opportunities await you with this pricing model.

1.4.3 Pay-per-use opportunities

The AWS pay-per-use pricing model creates new opportunities. You no longer need to make upfront investments in infrastructure. You can start servers on demand and only pay per hour of usage; and you can stop using those servers whenever you like and no longer have to pay for them. You don't need to make an upfront commitment regarding how much storage you'll use.

A big server costs exactly as much as two smaller ones with the same capacity. Thus you can divide your systems into smaller parts, because the cost is the same. This makes fault tolerance affordable not only for big companies but also for smaller budgets.

1.5 Comparing alternatives

AWS isn't the only cloud computing provider. Microsoft and Google have cloud offerings as well.

OpenStack is different because it's open source and developed by more than 200 companies including IBM, HP, and Rackspace. Each of these companies uses OpenStack to operate its own cloud offerings, sometimes with closed source add-ons. You could run your own cloud based on OpenStack, but you would lose most of the benefits outlined in section 1.3.

Comparing cloud providers isn't easy, because open standards are mostly missing. Functionality like virtual networks and message queuing are realized differently. If you

know what features you need, you can compare the details and make your decision. Otherwise, AWS is your best bet because the chances are highest that you'll find a solution for your problem.

Following are some common features of cloud providers:

- Virtual servers (Linux and Windows)
- Object store
- Load balancer
- Message queuing
- Graphical user interface
- Command-line interface

The more interesting question is, how do cloud providers differ? Table 1.2 compares AWS, Azure, Google Cloud Platform, and OpenStack.

Table 1.2 Differences between AWS, Microsoft Azure, Google Cloud Platform, and OpenStack

	AWS	Azure	Google Cloud Platform	OpenStack
Number of services	Most	Many	Enough	Few
Number of locations (multiple data centers per location)	9	13	3	Yes (depends on the OpenStack provider)
Compliance	Common standards (ISO 27001, HIPAA, FedRAMP, SOC), IT Grundschutz (Germany), G-Cloud (UK)	Common standards (ISO 27001, HIPAA, FedRAMP, SOC), ISO 27018 (cloud privacy), G-Cloud (UK)	Common standards (ISO 27001, HIPAA, FedRAMP, SOC)	Yes (depends on the OpenStack provider)
SDK languages	Android, Browsers (JavaScript), iOS, Java, .NET, Node.js (JavaScript), PHP, Python, Ruby, Go	Android, iOS, Java, .NET, Node.js (JavaScript), PHP, Python, Ruby	Java, Browsers (JavaScript), .NET, PHP, Python	-
Integration into development process	Medium, not linked to specific ecosystems	High, linked to the Microsoft ecosystem (for example, .NET development)	High, linked to the Google ecosystem (for example, Android)	-
Block-level storage (attached via network)	Yes	Yes (can be used by multiple virtual servers simultaneously)	No	Yes (can be used by multiple virtual servers simultaneously)
Relational database	Yes (MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server)	Yes (Azure SQL Database, Microsoft SQL Server)	Yes (MySQL)	Yes (depends on the OpenStack provider)
NoSQL database	Yes (proprietary)	Yes (proprietary)	Yes (proprietary)	No
DNS	Yes	No	Yes	No

Table 1.2 Differences between AWS, Microsoft Azure, Google Cloud Platform, and OpenStack *(continued)*

	AWS	Azure	Google Cloud Platform	OpenStack
Virtual network	Yes	Yes	No	Yes
Pub/sub messaging	Yes (proprietary, JMS library available)	Yes (proprietary)	Yes (proprietary)	No
Machine-learning tools	Yes	Yes	Yes	No
Deployment tools	Yes	Yes	Yes	No
On-premises data-center integration	Yes	Yes	Yes	No

In our opinion, AWS is the most mature cloud platform available at the moment.

1.6 Exploring AWS services

Hardware for computing, storing, and networking is the foundation of the AWS cloud. AWS runs software services on top of the hardware to provide the cloud, as shown in figure 1.9. A web interface, the API, acts as an interface between AWS services and your applications.

You can manage services by sending requests to the API manually via a GUI or programmatically via a SDK. To do so, you can use a tool like the Management Console, a web-based user interface, or a command-line tool. Virtual servers have a peculiarity: you can connect to virtual servers through SSH, for example, and gain administrator

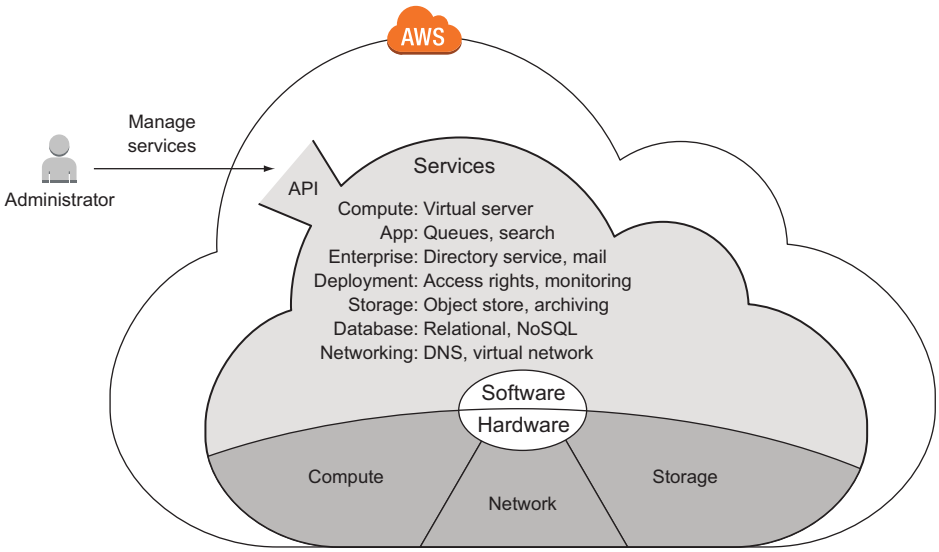


Figure 1.9 The AWS cloud is composed of hardware and software services accessible via an API.

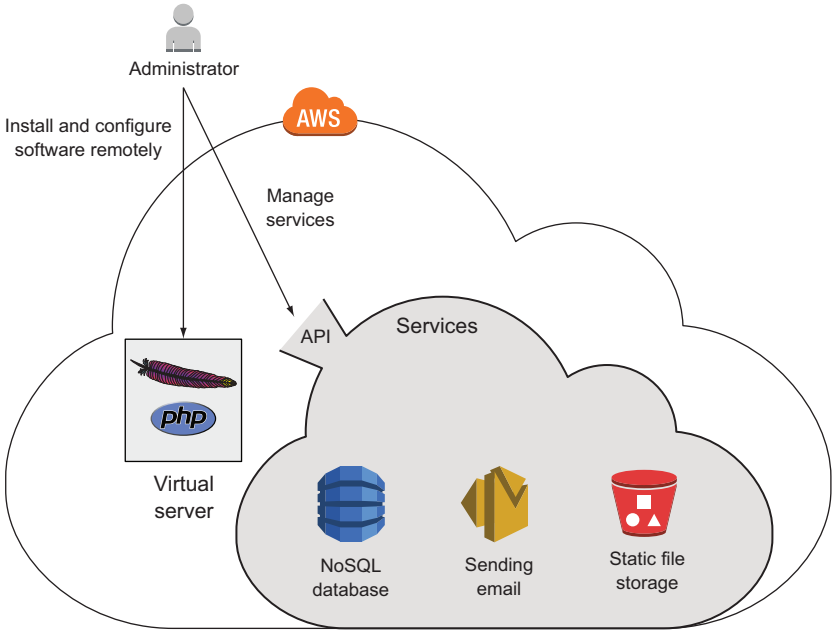


Figure 1.10 Managing a custom application running on a virtual server and dependent services

access. This means you can install any software you like on a virtual server. Other services, like the NoSQL database service, offer their features through an API and hide everything that's going on behind the scenes. Figure 1.10 shows an administrator installing a custom PHP web application on a virtual server and managing dependent services such as a NoSQL database used by the PHP web application.

Users send HTTP requests to a virtual server. A web server is installed on this virtual server along with a custom PHP web application. The web application needs to talk to AWS services in order to answer HTTP requests from users. For example, the web application needs to query data from a NoSQL database, store static files, and send email. Communication between the web application and AWS services is handled by the API, as figure 1.11 shows.

The number of different services available can be scary at the outset. The following categorization of AWS services will help you to find your way through the jungle:

- *Compute services* offer computing power and memory. You can start virtual servers and use them to run your applications.
- *App services* offer solutions for common use cases like message queues, topics, and searching large amounts of data to integrate into your applications.
- *Enterprise services* offer independent solutions such as mail servers and directory services.

- *Deployment and administration services* work on top of the services mentioned so far. They help you grant and revoke access to cloud resources, monitor your virtual servers, and deploy applications.
- *Storage* is needed to collect, persist, and archive data. AWS offers different storage options: an object store or a network-attached storage solution for use with virtual servers.
- *Database storage* has some advantages over simple storage solutions when you need to manage structured data. AWS offers solutions for relational and NoSQL databases.
- *Networking services* are an elementary part of AWS. You can define private networks and use a well-integrated DNS.

Be aware that we cover only the most important categories and services here. Other services are available, and you can also run your own applications on AWS.

Now that we've looked at AWS services in detail, it's time for you to learn how to interact with those services.

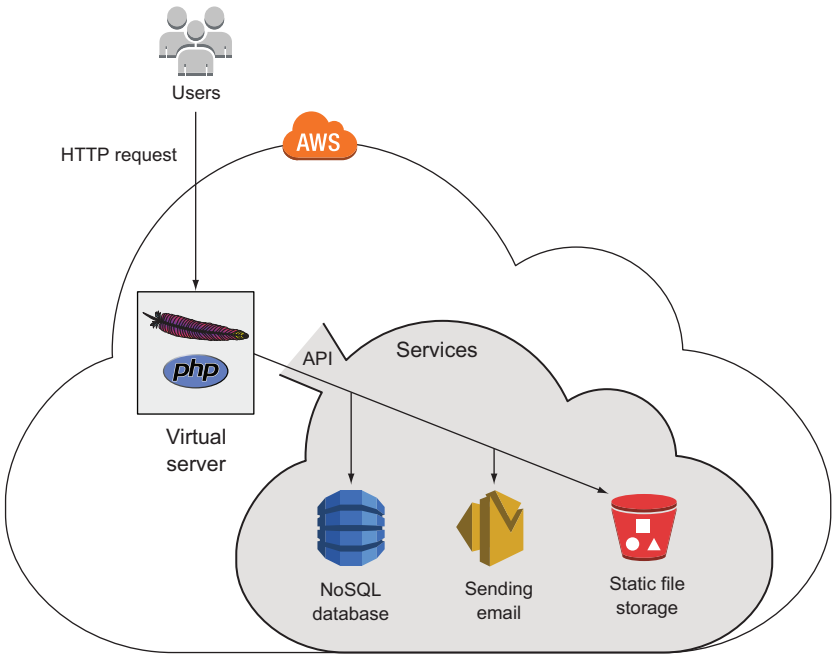


Figure 1.11 Handling an HTTP request with a custom web application using additional AWS services

1.7 Interacting with AWS

When you interact with AWS to configure or use services, you make calls to the API. The API is the entry point to AWS, as figure 1.12 demonstrates.

Next, we'll give you an overview of the tools available to make calls to the AWS API. You can compare the ability of these tools to automate your daily tasks.

1.7.1 Management Console

You can use the web-based Management Console to interact with AWS. You can manually control AWS with this convenient GUI, which runs in every modern web browser (Chrome, Firefox, Safari ≥ 5, IE ≥ 9); see figure 1.13.

If you're experimenting with AWS, the Management Console is the best place to start. It helps you to gain an overview of the different services and achieve success quickly. The Management Console is also a good way to set up a cloud infrastructure for development and testing.

1.7.2 Command-line interface

You can start a virtual server, create storage, and send email from the command line. With the command-line interface (CLI), you can control everything on AWS; see figure 1.14.

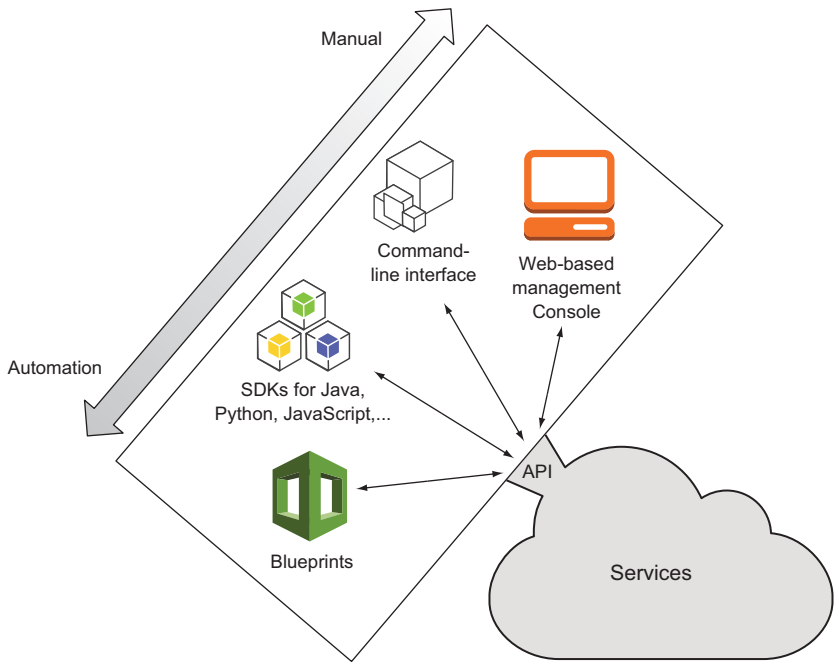


Figure 1.12 Tools to interact with the AWS API

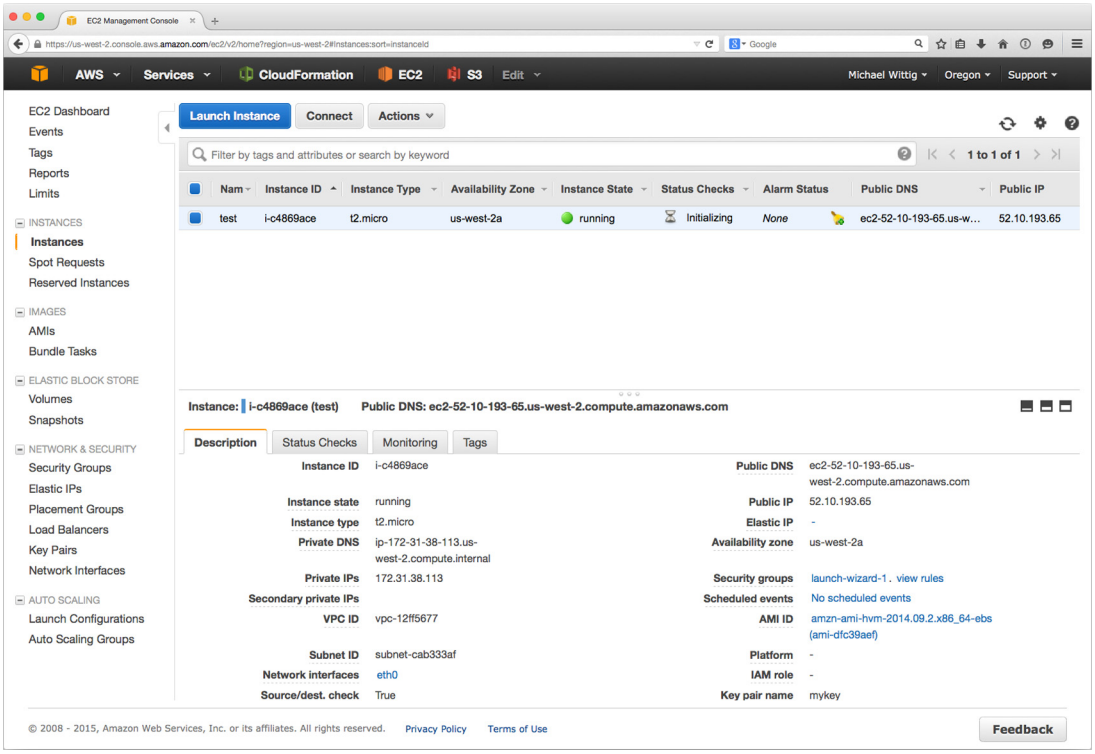


Figure 1.13 Management Console

```
michael — bash — 92×39
Last login: Fri Feb 20 09:32:45 on ttys000
mwittig:~ michael$ aws cloudwatch list-metrics --namespac "AWS/EC2" --max-items 3
{
    "Metrics": [
        {
            "Namespace": "AWS/EC2",
            "Dimensions": [
                {
                    "Name": "InstanceId",
                    "Value": "i-ed62dc0b"
                }
            ],
            "MetricName": "StatusCheckFailed_Instance"
        },
        {
            "Namespace": "AWS/EC2",
            "Dimensions": [
                {
                    "Name": "InstanceId",
                    "Value": "i-ed62dc0b"
                }
            ],
            "MetricName": "StatusCheckFailed"
        },
        {
            "Namespace": "AWS/EC2",
            "Dimensions": [
                {
                    "Name": "InstanceId",
                    "Value": "i-0a02beec"
                }
            ],
            "MetricName": "CPUUtilization"
        }
    ],
    "NextToken": "None___3"
}
mwittig:~ michael$
```

Figure 1.14 Command-line interface

The CLI is typically used to automate tasks on AWS. If you want to automate parts of your infrastructure with the help of a continuous integration server like Jenkins, the CLI is the right tool for the job. The CLI offers a convenient way to access the API and combine multiple calls into a script.

You can even begin to automate your infrastructure with scripts by chaining multiple CLI calls together. The CLI is available for Windows, Mac, and Linux, and there's also a PowerShell version available.

1.7.3 SDKs

Sometimes you need to call AWS from within your application. With SDKs, you can use your favorite programming language to integrate AWS into your application logic. AWS provides SDKs for the following:

- Android
- Browsers (JavaScript)
- iOS
- Java
- .NET
- Node.js (JavaScript)
- PHP
- Python
- Ruby
- Go

SDKs are typically used to integrate AWS services into applications. If you're doing software development and want to integrate an AWS service like a NoSQL database or a push-notification service, an SDK is the right choice for the job. Some services, such as queues and topics, must be used with an SDK in your application.

1.7.4 Blueprints

A *blueprint* is a description of your system containing all services and dependencies. The blueprint doesn't say anything about the necessary steps or the order to achieve the described system. Figure 1.15 shows how a blueprint is transferred into a running system.

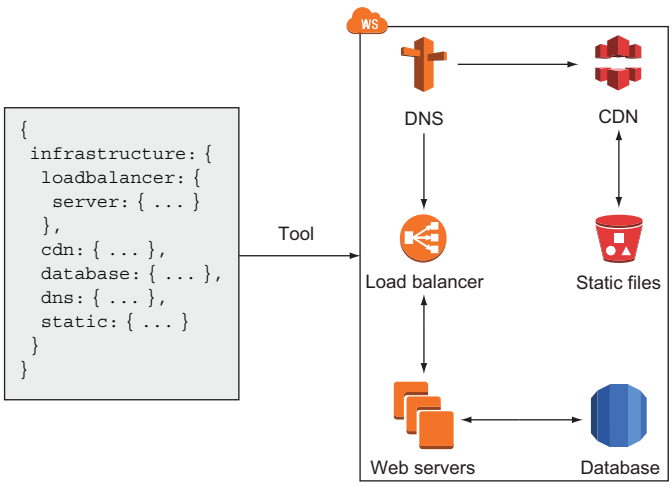


Figure 1.15 Infrastructure automation with blueprints

Consider using blueprints if you have to control many or complex environments. Blueprints will help you to automate the configuration of your infrastructure in the cloud. You can use blueprints to set up virtual networks and launch different servers into that network, for example.

A blueprint removes much of the burden from you because you no longer need to worry about dependencies during system creation—the blueprint automates the entire process. You'll learn more about automating your infrastructure in chapter 4.

It's time to get started creating your AWS account and exploring AWS practice after all that theory.

1.8 Creating an AWS account

Before you can start using AWS, you need to create an account. An AWS account is a basket for all the resources you own. You can attach multiple users to an account if multiple humans need access to the account; by default, your account will have one root user. To create an account, you need the following:

- A telephone number to validate your identity
- A credit card to pay your bills

> **Using an old account?**
>
> You can use your existing AWS account while working on the examples in this book. In this case, your usage may not be covered by the Free Tier, and you may have to pay for your usage.
>
> Also, if you created your existing AWS account before December 4, 2013, you should create a new one: there are legacy issues that may cause trouble when you try our examples.

1.8.1 Signing up

The sign-up process consists of five steps:

1. Provide your login credentials.
2. Provide your contact information.
3. Provide your payment details.
4. Verify your identity.
5. Choose your support plan.

Point your favorite modern web browser to https://aws.amazon.com, and click the Create a Free Account / Create an AWS Account button.

1. Providing your login credentials

The Sign Up page, shown in figure 1.16, gives you two choices. You can either create an account using your Amazon.com account or create an account from scratch. If you create the account from scratch, follow along. Otherwise, skip to step 5.

Fill in your email address, and select I Am a New User. Go on to the next step to create your login credentials. We advise you to choose a strong password to prevent misuse

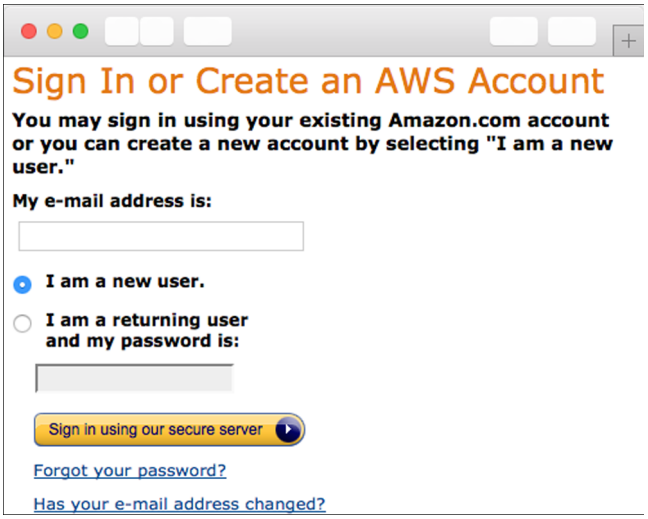


Figure 1.16 Creating an AWS account: Sign Up page

of your account. We suggest a password with 16 characters, numbers, and symbols. If someone gets access to your account, they can destroy your systems or steal your data.

2. PROVIDING YOUR CONTACT INFORMATION

The next step, as shown in figure 1.17, is to provide your contact information. Fill in all the required fields, and continue.

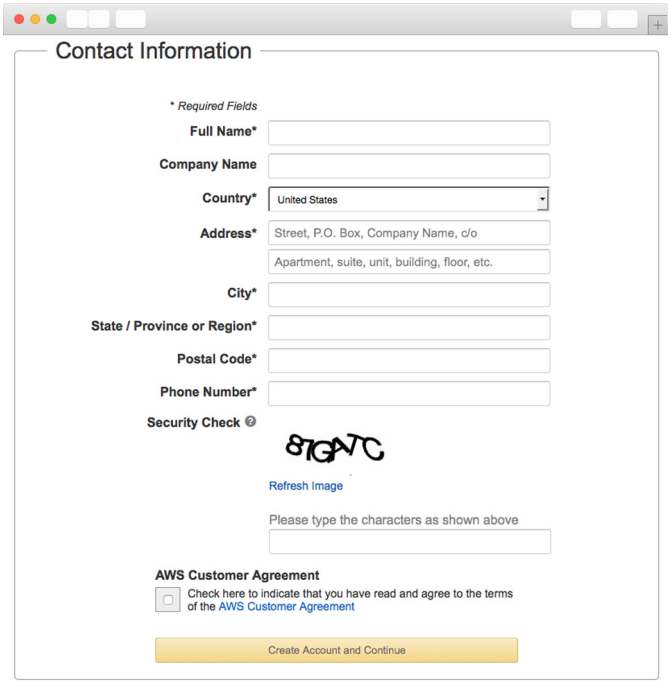


Figure 1.17 Creating an AWS account: providing your contact information

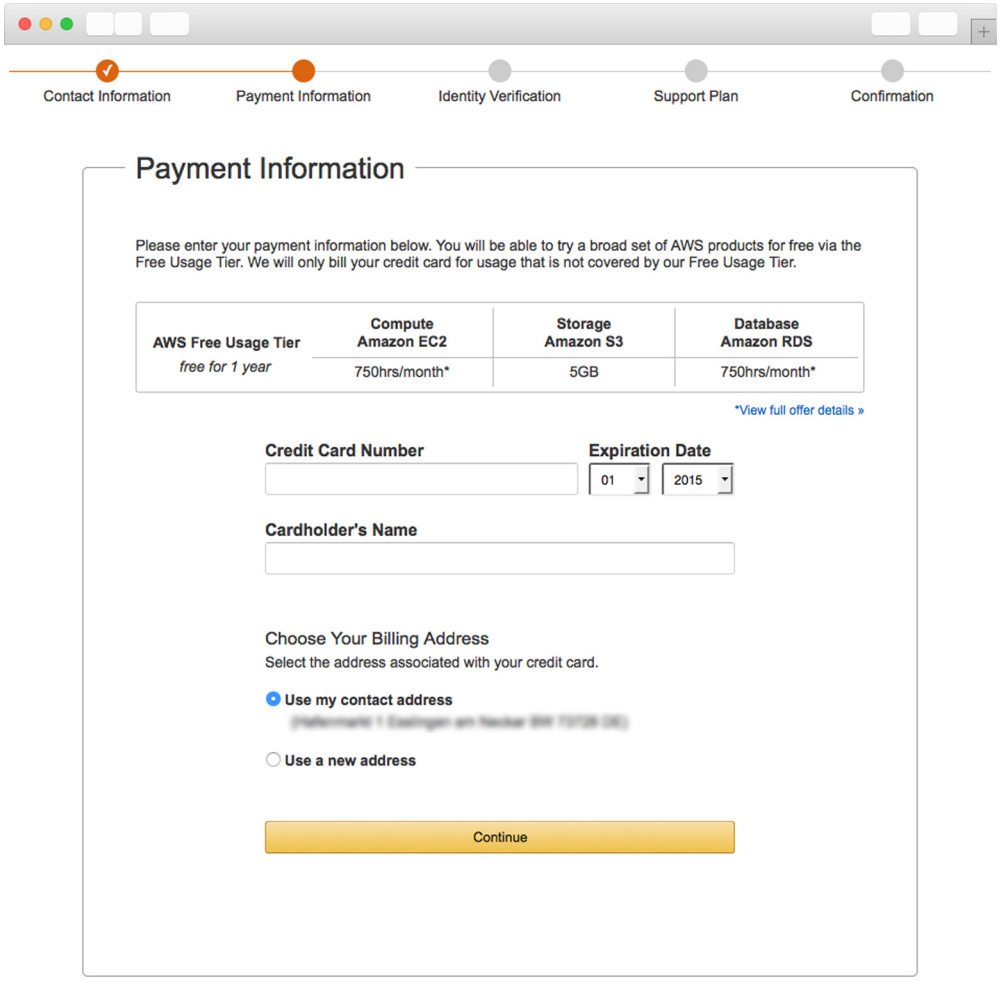


Figure 1.18 Creating an AWS account: providing your payment details

3. Provide your payment details

Now the screen shown in figure 1.18 asks for your payment information. AWS supports MasterCard and Visa. You can set your preferred payment currency later, if you don't want to pay your bills in USD; supported currencies are EUR, GBP, CHF, AUD, and some others.

4. Verifying your identity

The next step is to verify your identity. Figure 1.19 shows the first step of the process.

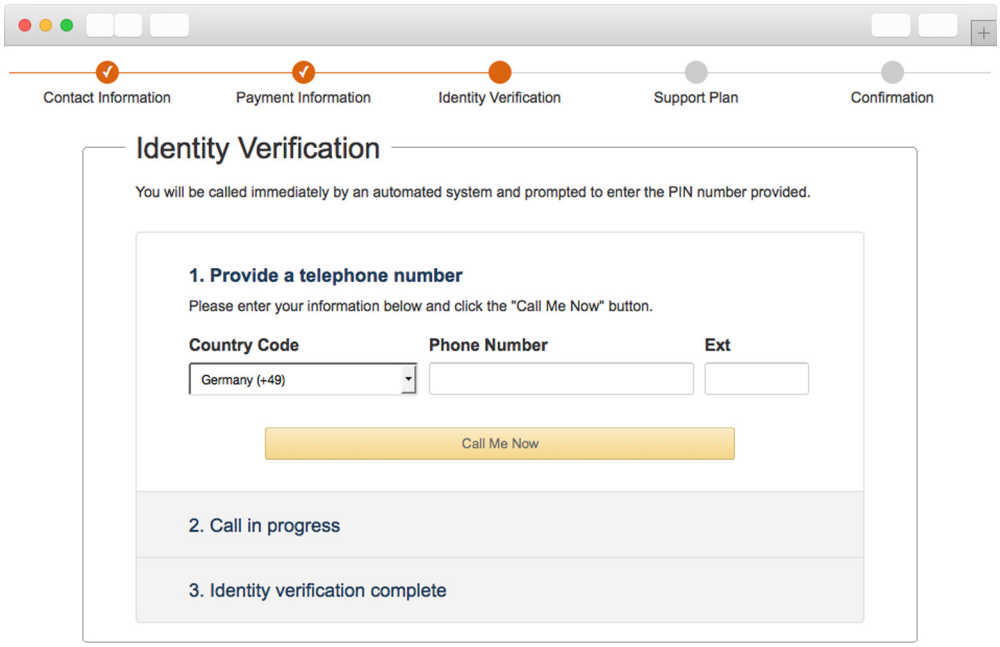


Figure 1.19 Creating an AWS account: verifying your identity (1 of 2)

After you complete the first part, you'll receive a call from AWS. A robot voice will ask you for your PIN, which will be like the one shown in figure 1.20. Your identity will be verified, and you can continue with the last step.

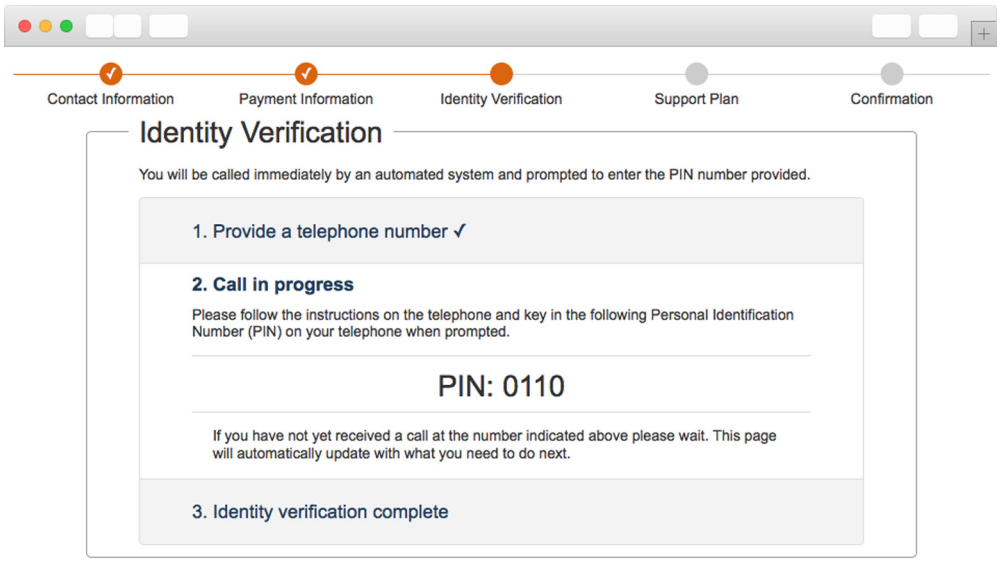


Figure 1.20 Creating an AWS account: verifying your identity (2 of 2)

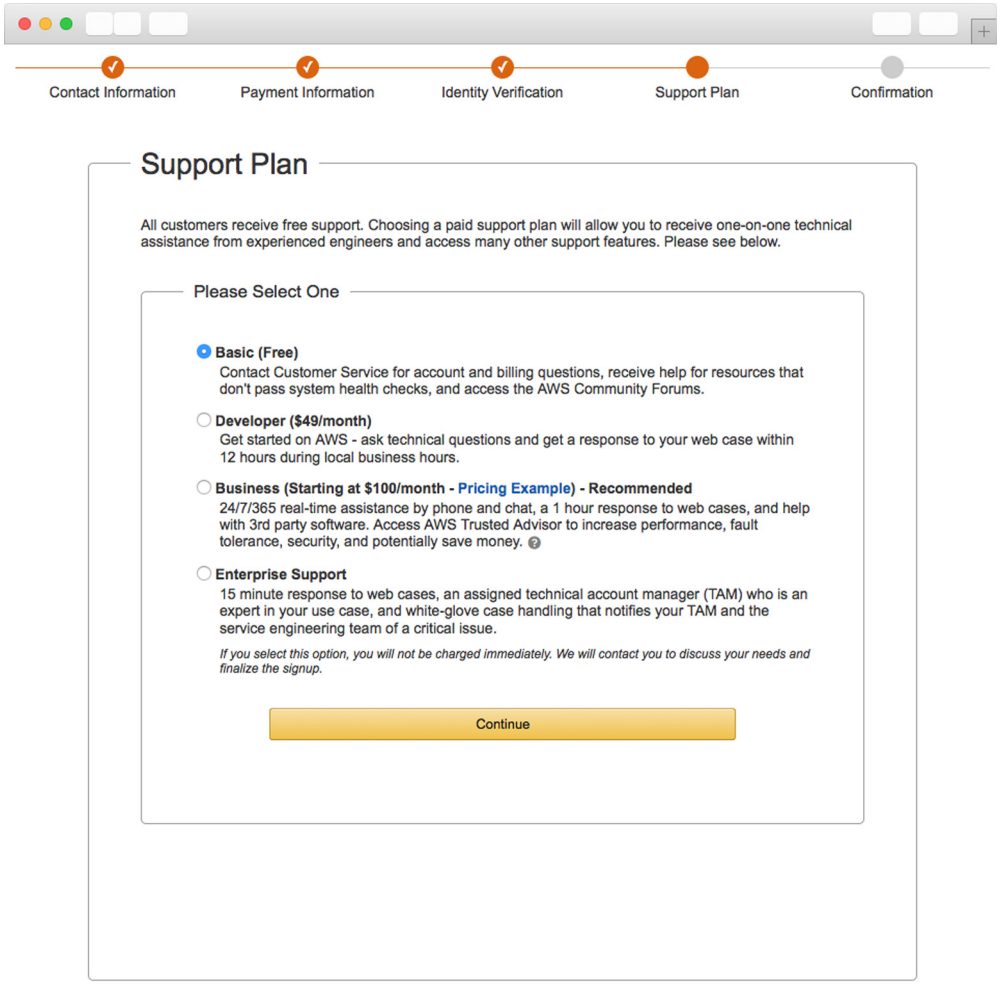


Figure 1.21 Creating an AWS account: choosing your support plan

5. Choosing your support plan

The last step is to choose a support plan; see figure 1.21. In this case, select the Basic plan, which is free. If you later create an AWS account for your business, we recommend the Business support plan. You can even switch support plans later.

High five! You're done. Now you can log in to your account with the AWS Management Console.

1.8.2 Signing In

You have an AWS account and are ready to sign in to the AWS Management Console at https://console.aws.amazon.com. As mentioned earlier, the Management Console is a web-based tool you can use to control AWS resources. The Management Console

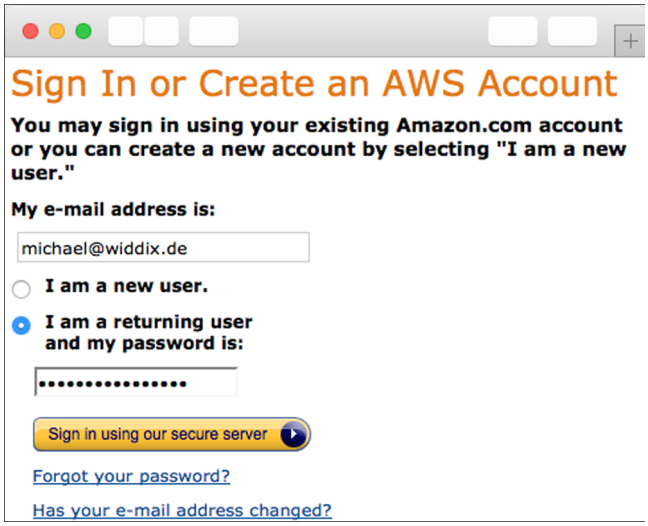


Figure 1.22 Sign in to the Management Console.

uses the AWS API to make most of the functionality available to you. Figure 1.22 shows the Sign In page.

Enter your login credentials and click Sign In Using Our Secure Server to see the Management Console, shown in figure 1.23.

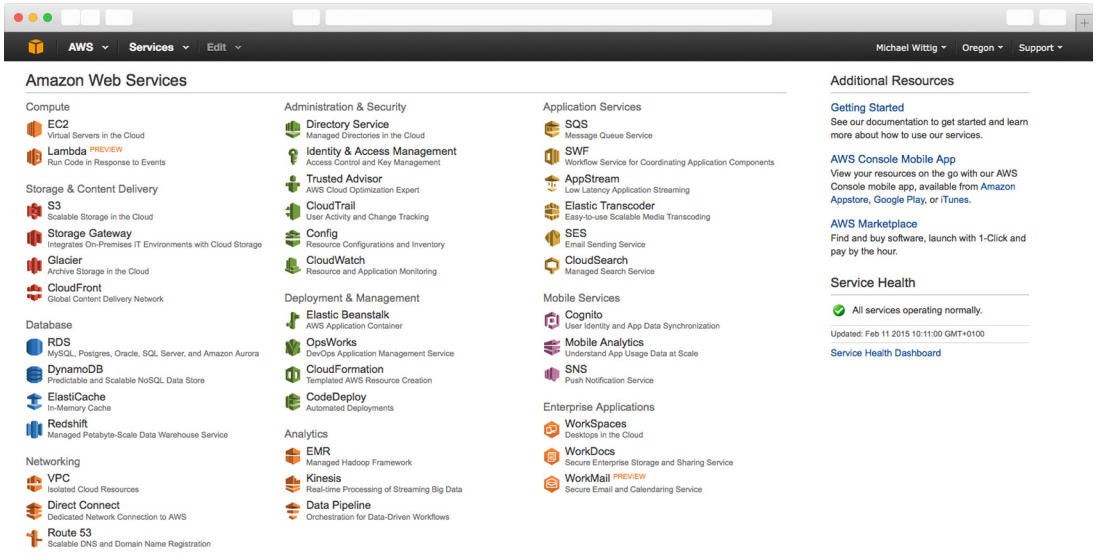


Figure 1.23 AWS Management Console

The most important part is the navigation bar at the top; see figure 1.24. It consists of six sections:

- *AWS*—Gives you a fast overview of all resources in your account.
- *Services*—Provides access to all AWS services.
- *Custom section (Edit)*—Click Edit and drag-and-drop important services here to personalize the navigation bar.
- *Your name*—Lets you access billing information and your account, and also lets you sign out.
- *Your region*—Lets you choose your region. You'll learn about regions in section 3.5. You don't need to change anything here now.
- *Support*—Gives you access to forums, documentation, and a ticket system.

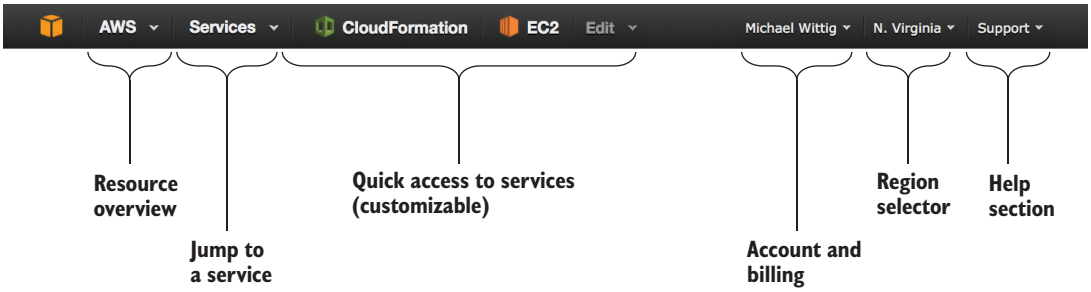


Figure 1.24 AWS Management Console navigation bar

Next, you'll create a key pair so you can connect to your virtual servers.

1.8.3 Creating a key pair

To access a virtual server in AWS, you need a *key pair* consisting of a private key and a public key. The public key will be uploaded to AWS and inserted into the virtual server. The private key is yours; it's like your password, but much more secure. Protect your private key as if it's a password. It's your secret, so don't lose it—you can't retrieve it.

To access a Linux server, you use the SSH protocol; you'll authenticate with the help of your key pair instead of a password during login. You access a Windows server via Remote Desktop Protocol (RDP); you'll need your key pair to decrypt the administrator password before you can log in.

The following steps will guide you to the dashboard of the EC2 service, which offers virtual servers, and where you can obtain a key pair:

1. Open the AWS Management Console at https://console.aws.amazon.com.
2. Click Services in the navigation bar, find the EC2 service, and click it.
3. Your browser shows the EC2 Management Console.

The EC2 Management Console, shown in figure 1.25, is split into three columns. The first column is the EC2 navigation bar; because EC2 is one of the oldest services, it has many

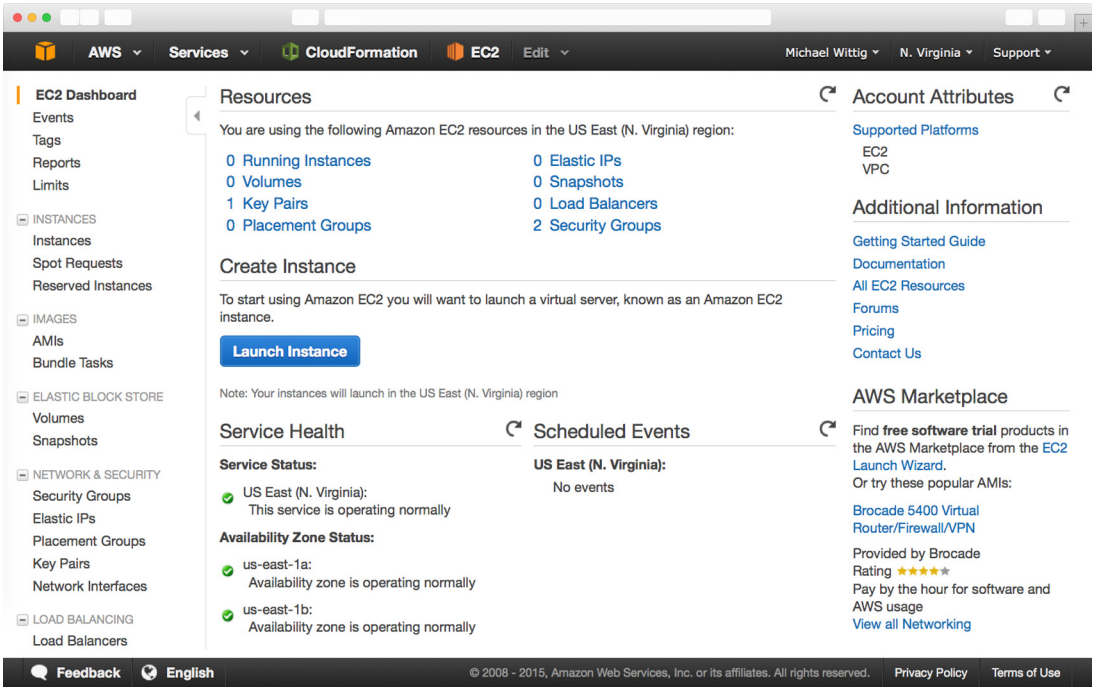


Figure 1.25 EC2 Management Console

features that you can access via the navigation bar. The second column gives you a brief overview of all your EC2 resources. The third column provides additional information.

Follow these steps to create a new key pair:

1. Click Key Pairs in the navigation bar under Network & Security.
2. Click the Create Key Pair button on the page shown in figure 1.26.
3. Name the Key Pair `mykey`. If you choose another name, you must replace the name in all the following examples!

During key-pair creation, you downloaded a file called mykey.pem. You must now prepare that key for future use. Depending on your operating system, you may need to do things differently, so please read the section that fits your OS.

Using your own key pair

It's also possible to upload the public key part from an existing key pair to AWS. Doing so has two advantages:

- You can reuse an existing key pair.
- You can be sure that only you know the private key part of the key pair. If you use the Create Key Pair button, AWS knows (at least briefly) your private key.

We decided against that approach in this case because it's less convenient to implement in a book.

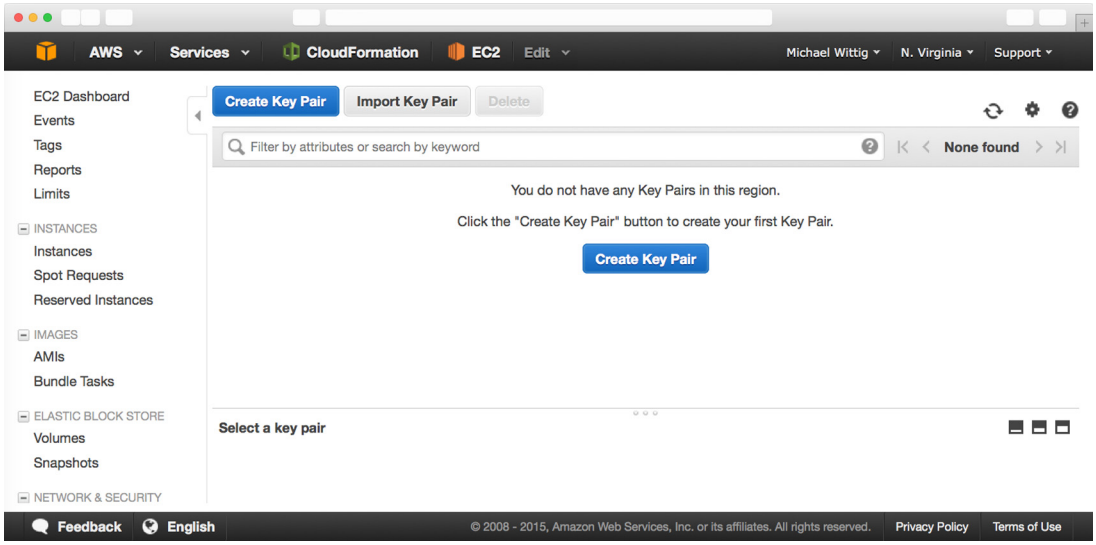


Figure 1.26 EC2 Management Console key pairs

Linux and Mac OS X

The only thing you need to do is change the access rights of mykey.pem so that only you can read the file. To do so, run `chmod 400 mykey.pem` in the terminal. You'll learn about how to use your key when you need to log in to a virtual server for the first time in this book.

Windows

Windows doesn't ship a SSH client, so you need to download the PuTTY installer for Windows from http://mng.bz/A1bY and install PuTTY. PuTTY comes with a tool called PuTTYgen that can convert the mykey.pem file into a mykey.ppk file, which you'll need:

1. Run the application PuTTYgen. The screen shown in figure 1.27 opens.
2. Select SSH-2 RSA under Type of Key to Generate.
3. Click Load.
4. Because PuTTYgen displays only *.pkk files, you need to switch the file extension of the File Name field to All Files.
5. Select the mykey.pem file, and click Open.
6. Confirm the dialog box.
7. Change Key Comment to `mykey`.
8. Click Save Private Key. Ignore the warning about saving the key without a passphrase.

Your .pem file is now converted to the .pkk format needed by PuTTY. You'll learn how to use your key when you need to log in to a virtual server for the first time in this book.

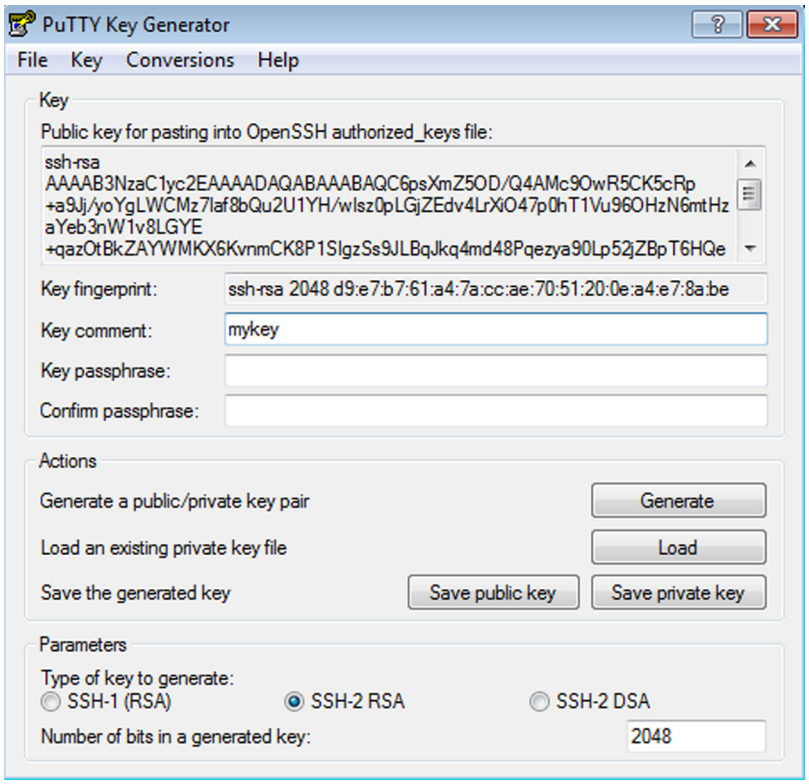


Figure 1.27 PuTTYgen allows you to convert the downloaded .pem file into the .pkk file format needed by PuTTY.

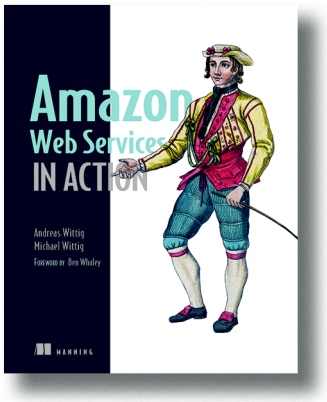
1.8.4 Creating a billing alarm

Before you use your AWS account in the next chapter, we advise you to create a billing alarm. If you exceed the Free Tier, an email is sent to you. The book warns you whenever an example isn't covered by the Free Tier. Please make sure that you carefully follow the cleanup steps after each example. To make sure you haven't missed something during cleanup, please create a billing alarm as advised by AWS: http://mng.bz/M7Sj.

1.9 Summary

- Amazon Web Services (AWS) is a platform of web services offering solutions for computing, storing, and networking that work well together.
- Cost savings aren't the only benefit of using AWS. You'll also profit from an innovative and fast-growing platform with flexible capacity, fault-tolerant services, and a worldwide infrastructure.
- Any use case can be implemented on AWS, whether it's a widely used web application or a specialized enterprise application with an advanced networking setup.

- You can interact with AWS in many different ways. You can control the different services by using the web-based GUI; use code to manage AWS programmatically from the command line or SDKs; or use blueprints to set up, modify, or delete your infrastructure on AWS.
- Pay-per-use is the pricing model for AWS services. Computing power, storage, and networking services are billed similarly to electricity.
- Creating an AWS account is easy. Now you know how to set up a key pair so you can log in to virtual servers for later use.



Physical data centers require lots of equipment and take time and resources to manage. If you need a data center, but don't want to build your own, Amazon Web Services may be your solution. Whether you're analyzing real-time data, building software as a service, or running an e-commerce site, AWS offers you a reliable cloud-based platform with services that scale.

Amazon Web Services in Action introduces you to computing, storing, and networking in the AWS cloud. The book will teach you about the most important services on AWS. You will also learn about best practices regarding security, high availability and scalability.You'll start with a broad overview of cloud computing and AWS and learn how to spin-up servers manually and from the command line. You'll learn how to automate your infrastructure by programmatically calling the AWS API to control every part of AWS. You will be introduced to the concept of Infrastructure as Code with the help of AWS CloudFormation.You will learn about different approaches to deploy applications on AWS. You'll also learn how to secure your infrastructure by isolating networks, controlling traffic and managing access to AWS resources. Next, you'll learn options and techniques for storing your data. You will experience how to integrate AWS services into your own applications by the use of SDKs. Finally, this book teaches you how to design for high availability, fault tolerance, and scalability.

What's inside

- Overview of AWS cloud concepts and best practices
- Manage servers on EC2 for cost-effectiveness
- Infrastructure automation with Infrastructure as Code (AWS CloudFormation)
- Deploy applications on AWS
- Store data on AWS: SQL, NoSQL, object storage and block storage
- Integrate Amazon's pre-built services
- Architect highly available and fault tolerant systems

Written for developers and DevOps engineers moving distributed applications to the AWS platform.

Implementing Security as a Service

As exciting and interesting as the new approaches to web development and security are, most developers also have to consider how to manage existing systems based on older technologies. This chapter covers SAML, WS-Trust, and other protocols and approaches that are now considered legacy systems. We felt it was important to provide a historical context to help you see how far things have come in the last 7 years.

Chapter 8 from *SOA Security* by Ramarao Kanneganti and Prasad A. Chodavarapu

Implementing security as a service

This chapter covers

- SAMLAssertions
- OpenSAML
- WS-Trust and SAML protocol

In part II, you saw some of the technological building blocks needed to implement security for web services: authentication, encryption, and signatures. If you are going to secure only a few simple services, what you have learned up to this point should hold you in good stead. For example, if you are an application developer simply seeking to secure the services offered by your back-end modules to your front-end modules, you already know enough to get your work done.

If you are developing or implementing an enterprise-class SOA security solution, there are a few more fundamental pieces that are needed to develop full-fledged frameworks, strategies, and architectures.[1] In particular, we must address

[1] Kerberos, described in chapter 5, can by itself provide the basis for an enterprise-class security framework. The use of Kerberos across trust domains (enterprises, or even divisions within enterprises) is rare. We need alternate security mechanisms that scale within and beyond an enterprise.

the security management issues that we described in the first chapter. To recap, enterprise SOA security solutions need to address the following concerns:

- *Ease of development* Are there ways we can reduce or eliminate the burden of security enforcement from developers of services and service consumers? If we can find such ways, the cost of developing a new service or service consumer can be brought down.
- *Manageability* How can we ensure consistent enforcement of security policies in all services and service consumers deployed within an enterprise? If a security decision is taken to allow access to a specific resource, how do we trace it to a security policy? Enterprise SOA security solutions need to provide easy-to-use mechanisms to answer these questions.[1]
- *Interoperability* How can we ensure interoperability between different security solutions if standards allow different options for good reasons?

In other words, enterprise SOA security solutions need to take the costs of development and management into account and ensure interoperability. In this chapter and the next we will show you how to do so.

In this chapter, we will present the idea of security as a service. To understand this idea, consider the following: One way of securing services is to implement security for each of them. Since the security is not dependent on the actual service, there will be many common elements in different implementations of security. If this were normal application development, we would extract this commonality into a library. For the reasons we explained in the first chapter, in SOA it is natural to extract security into a service, so that it can be used by any technology and platform.

Thankfully, there are standards that are developed specifically for addressing this need. These include WS-Addressing, SAML assertions, the SAML protocol, WS-Trust, and AON. Of these, we introduced WS-Addressing in chapter 3. We will describe SAML assertions, the SAML protocol, and WS-Trust in this chapter. AON will be described in appendix E. With the help of these standards, you will learn how to develop security as a separate service so that it can be used for SOA security.

We will start this chapter by introducing the idea of security as a service and how it is useful in securing SOA. Subsequently, we will introduce you to standards that allow security to be used as a service. We will present several use cases to understand how these standards can be used. Finally, we will show you how to implement a security service that uses these standards.

[1] These questions are significant for other reasons as well. Regulations (such as the Sarbanes-Oxley Act in the U.S.) and corporate governance policies require mechanisms that guarantee consistent enforcement of security policies.

8.1 Security as a service

In the first chapter, we discussed basic security issues. As shown in table 8.1, we've already addressed them except protection against attacks and privacy. Since we will refer to these technologies later in this chapter, let's summarize.

Table 8.1 Review of security technologies described in earlier chapters

Requirement	Technology choices	See chapter
Making and verifying identity claims	Username and password	4
	Username and password digest	4
	Kerberos	5
	Digital signatures	7
	Authentication using JAAS against a variety of repositories	4
Protecting data confidentiality	Point-to-point secure transport with SSL	6
	Selective encryption with shared secrets, PKI, or Kerberos	5 and 6
Verifying data integrity and nonrepudiation	Point-to-point secure transport with SSL	6
	Selective signing with PKI or Kerberos	5, 6, and 7

If all you have are a few simple services, you can build one or more of these mechanisms into each of the service and service consumer implementations. Figure 8.1 depicts this approach.

This approach to securing services is simple to implement. The standard protocols and established practices in security are sufficient to deliver this solution. Consequently, it is easy to understand and develop. In fact, most enterprise applications are built this way even today.

But what if you need to secure a large number of services, as would be the case in any enterprise? Would you still use the same approach?

Consider what happens if you need to secure a large number of services. If we go by the approach shown in figure 8.1, we have to replicate the security enforcement machinery across all services and service consumers. Worse still, if security requirements differ for each application (for example, some services may have stricter security controls than others) then the security machinery in each will end up looking similar with subtle differences, leading to high maintenance costs. In addition, this simple design approach does not lend itself to more advanced use cases. What if a service needs to contact other services when processing a message? It would need to transfer the entire security context to all the service providers, which can be complex.

If you are planning on building an enterprise-class framework for securing a large number of services, you will want to explore ways of shifting at least some of the security enforcement burden from services and service consumers to a shared security ser-

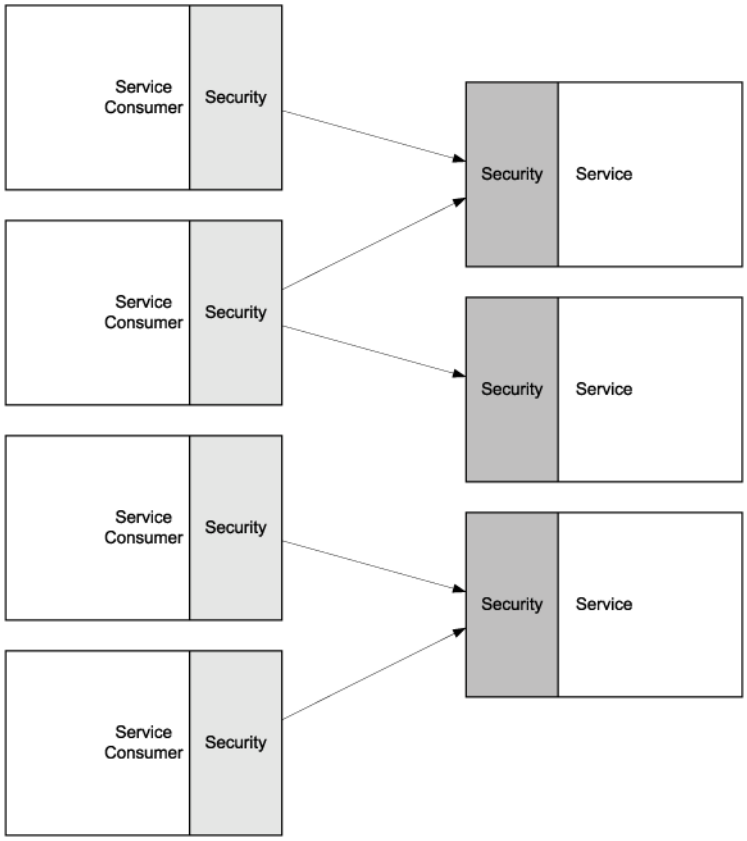


Figure 8.1 Security implemented as part of each service and service consumer. Each service implements its own security, which is invoked as a part of service consumer and provider. For instance, the consumer may add username and password information and the provider may validate them and grant access

vice. A shared security service will help you enforce security policies consistently across all services. Figure 8.2 illustrates this approach.

One may argue that separation of security as a shared service is not really necessary to ensure reuse of security machinery; one can always offer the security implementations as libraries that all services and service consumers can reuse. Even though a security service does offer a superior reuse mechanism—one that is independent of programming languages and platforms—note that reuse is not the main reason why we are considering implementing security as a shared service. What we are really seeking to address is the challenge of deploying, managing, and evolving security enforcement mechanisms across a large number of services. A security service can be centrally managed and modified quickly to meet rapidly changing business needs. Security machinery reused via libraries cannot provide the same benefit.

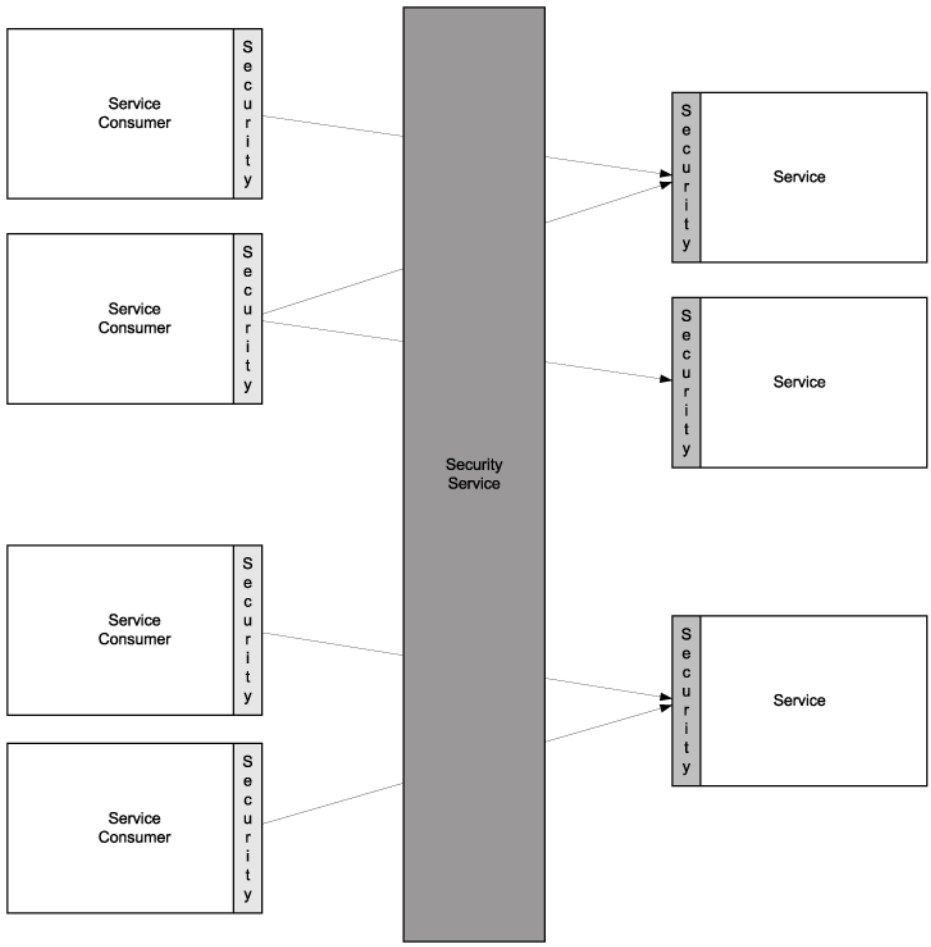


Figure 8.2 Security implemented as a separate service to offload most of the security enforcement burden from services and service consumers. Note that services and service consumers may still handle some security tasks. For example, they may have to understand how to use the security service.

It's one thing to make a case for a shared security service and another thing to implement it. We have to see how we can pull it off. For example, we need to figure out how services and service consumers invoke the security service, how the credentials are communicated, and so on.

We'll discuss the technical feasibility of a security service next. Before we do that, a note on terminology is in order. For reasons that will soon become obvious, we will hereafter stick to the terms *source endpoint* and *destination endpoint* instead of using terms such as service/service consumer, client/server, and sender/receiver.

8.1.1 Is a security service technically feasible?

Implementing security as a service is a lot more technically challenging than enforcing security at the endpoints. For it to be technically feasible,[1] we need to address the following questions:

- *Who invokes the security service?* The issue of who invokes the security service seems simple: Whoever needs it will invoke it. What is not obvious is who is supposed to need it. We can say that the source endpoint needs to obtain a security clearance before contacting the destination endpoint. Or, we can say that the destination endpoint needs to validate the credentials submitted by the source endpoint. These choices mean that the security service needs to support different kinds of use cases. For example, a source may simply be interested in getting a service ticket or a destination endpoint may be interested in authenticating and authorizing a request.
- *How is the security context communicated to the destination endpoint?* Endpoints are often interested in the results of security enforcement. For example, a destination endpoint might be interested in knowing the authenticated identity of the caller. Another endpoint might need more than just the identity; it might be interested in knowing the privileges granted to the caller. As you can see, the result of security enforcement is a context that needs to be communicated to the destination endpoint. What constitutes the context varies based on the needs of the endpoint.
- *What is the interface for the security service?* Given the diverse set of use cases and security technologies a security service should provide, it is not easy to come up with an interface that serves all needs. For example, a security service should be able to validate identity claims made using username tokens, X.509 certificates, and Kerberos tickets. Not only that, the interface should allow for securing the communication with the security service itself!

These issues are addressed by various standards, which we will introduce next. Once we understand these standards, it will become clear that a security service is indeed a technically feasible idea.

8.1.2 Standards for implementing security as a service

A number of standards and technologies need to be brought together when implementing security as a service. We have already introduced some of these technologies in previous chapters. We will introduce the rest here. Table 8.2 lists all of these standards and technologies and provides a pointer to the chapter and section that describes each of them.

[1] We are not discussing the practical issues in migrating from endpoint-enforced security to a security service. We will discuss them in chapter 10.

Table 8.2 Standards and technologies that make implementation of a security service technically feasible

Standard/Technology	Description	Described in
WS-Addressing	Standardizes SOAP headers for preserving destination endpoint information when routing a message via the security service	3.5
Application-Oriented Networking (AON)	Technology that enables the network to understand application-level messages and even become a security service provider	3.5 and appendix E
Security Assertion Markup Language (SAML)	Provides the syntax for conveying the findings of a security service	8.3
WS-Trust	Describes interfaces for a security service that can issue, validate, renew, and cancel security tokens such as SAML assertions and Kerberos tickets	8.5.1
SAML protocol	Describes interfaces for a security service that returns its findings as SAML	8.5.2

Before we look at SAML, WS-Trust, and the SAML protocol, it helps to look at the possible use cases for a security service. The use case analysis in the next section will help you understand the technical issues in implementing security as a service. This understanding will in turn help you appreciate the motivations behind each of the standards and technologies we listed in table 8.2.

8.2 Analyzing possible uses of a security service

To understand how to create a security service, we have to first understand how it can be used. The possible uses of a security service can be classified into the following five use cases based on how the security service is invoked.

1. Destination endpoint invokes security service out-of-band
2. Source endpoint invokes security service out-of-band
3. Both endpoints invoke security service out-of-band
4. Messages are explicitly routed via the security service by the source endpoint or by a previous intermediary in the message path
5. A smart network device automatically routes messages via the security service

In this section, we will analyze each of these use cases. We will describe the scenario, identify the standards and technologies that can help in the implementation, and evaluate the pros and cons of invoking the security service as described. This exercise will help us identify the different pieces of the technology puzzle needed to create a security service.

Let's start our discussion with a use case in which the destination endpoint invokes the security service out-of-band.

8.2.1 Use case 1: Destination endpoint invokes security service out-of-band

In all the examples we have shown so far, with the exception of in chapter 5 where we discussed Kerberos, the destination endpoint bears the burden of security enforcement. So, the most natural candidate for invoking the security service is the destination endpoint itself. Instead of calling different library functions for executing security-related logic, the destination endpoint can invoke the security service. Let's call this use case 1.

In this use case, the source endpoint is not at all aware of the security service. The source endpoint simply invokes the destination endpoint, which in turn invokes the security service. As the security service is not in the message path, the security service is said to be invoked out-of-band. Figure 8.3 depicts this.

Figure 8.3 The destination endpoint invokes the security service out-of-band. The burden of security is moved to a separate service. Still, the destination endpoint (that is, the service provider) must invoke it explicitly.

You will recognize that this use case is quite commonly found if you look at an LDAP server as a security service. An LDAP server provides authentication as a service to server applications that wish to authenticate users contacting them. It also often provides server applications with the information necessary to make authorization decisions, but leaves the final authorization decision to the applications themselves.

The security service we wish to implement needs to provide significantly more functionality than a traditional LDAP server. For example, the security service may need to decide which resources the authenticated source endpoint is allowed to access and for how long. That is, the security service may need to provide authorization functionality, too.

The security service may also need to provide the destination endpoint with additional details about the source endpoint that the security service gets to know when doing authentication. For example, the security service may get to know the source endpoint's location and preferences during authentication. The destination endpoint may need such information for its business logic, consequently the security service will need to communicate this to the destination endpoint.

Furthermore, as we want to provide the security service even over unsecured networks, all communication with the security service needs to be protected. Otherwise, a man in the middle may be able compromise security.

RELEVANT STANDARDS AND TECHNOLOGIES

Of the standards and technologies described in table 8.2, the following are relevant in this use case:

- *SAML* The security service can express its findings to the destination endpoint in the form of standard SAML assertions.
- *WS-Trust and/or the SAML protocol* These standards specify how the security service can be invoked.

WS-Addressing and AON are not relevant in this use case, as messages are not routed via the security service.

PROS AND CONS

There are several advantages to the destination endpoint invoking the security service. It is only a small step from the way we have been implementing security. Source endpoints need not even know about the existence of the security service, and need not understand how to interact with one.

The big disadvantage is that every destination endpoint needs to know how to interact with the security service. Furthermore, every destination endpoint is forced to spend time and effort obtaining the security decision. As the number of service requests increases, the load on the destination endpoint increases.

From the perspective of source endpoints, too, this use case has problems. The source endpoints are forced to reveal their credentials to the destination endpoint. This allows destination endpoints to steal and reuse the source's credentials for contacting other services. (We discussed this service-provider abuse in chapter 5.)

The obvious alternative to the destination endpoint invoking the security service is to leave that burden to the source endpoint. Let's discuss that possibility next.

8.2.2 Use case 2: Source endpoint invokes security service out-of-band

In this use case, the source endpoint invokes the security service to get a security token that it in turn submits to the destination endpoint. At the destination endpoint, the security token is examined and appropriate action is taken. Figure 8.4 illustrates this use case.

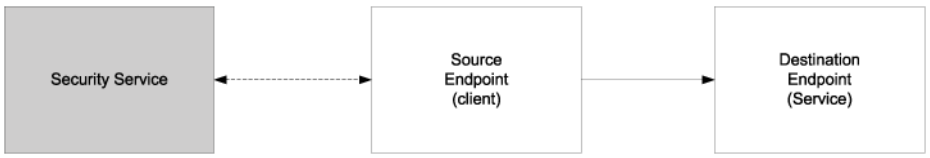


Figure 8.4 The source endpoint invokes the security service out-of-band. Just as before, security is handled by the separate security service. The burden of invoking it falls to the source endpoint; i.e. the consumer. Since the result should be accepted by the destination endpoint, the security service needs to implement standards such as SAML.

Kerberos operates in a similar fashion. There, too, the client gets a token—a service ticket—which is submitted to the service. The ticket can carry the complete security context in encrypted and signed form. Refer to chapter 5 for an introduction to Kerberos.

RELEVANT STANDARDS AND TECHNOLOGIES

Of the standards and technologies described in table 8.2, the following are relevant in this use case.

- *SAML* The security token returned by the security service can take the form of a standard SAML assertion.
- *WS-Trust and/or the SAML protocol* These standards specify how the security service can be invoked.

As in use case 1, WS-Addressing and AON are not relevant, as messages are not routed via the security service

PROS AND CONS

This use case has a clear advantage over the previous one: It spreads the burden of invoking the security service over the source endpoints. So, this use case scales better than the previous one. In addition, this solution does not reveal the security credentials to the destination endpoint. This prevents the destination endpoint owner from repurposing the submitted credentials to access a different service.

But there are some disadvantages with this use case. Any source endpoint wishing to use the destination endpoint needs to invoke the security service. This increases the programming complexity on the part of the clients, reducing the usability of the service.

In addition, the entire security context that the destination endpoint needs should be available in the ticket. Since the source endpoint cannot be sure what information is required, it will have to carry all the potentially required information in the ticket. This can get bulky as more and more attributes are added to the security context.

One way to take care of the problem of having to carry the entire security context (rather than just the necessary data) to the destination endpoint is to let both endpoints talk to the security service. That possibility is use case 3, which we discuss next.

8.2.3 Use case 3: Both endpoints invoke security service out-of-band

In this scenario, the source endpoint talks to the security service and obtains a security token that provides a partial security context, enough to provide most common information required by any service. If a destination endpoint receives a security token and requires further information about the source endpoint, it will turn to the security service to obtain that information. Figure 8.5 illustrates this possibility.

RELEVANT STANDARDS AND TECHNOLOGIES

Of the standards and technologies described in table 8.2, the following are relevant in this use case.

- *SAML* The security token returned by the security service to the source endpoint can take the form of a standard SAML assertion. Similarly, when the destination endpoint queries the security service for more information on a source endpoint, the security service's response can be in the form of a standard SAML assertion.
- *WS-Trust and/or the SAML protocol* These standards specify how the security service can be invoked.

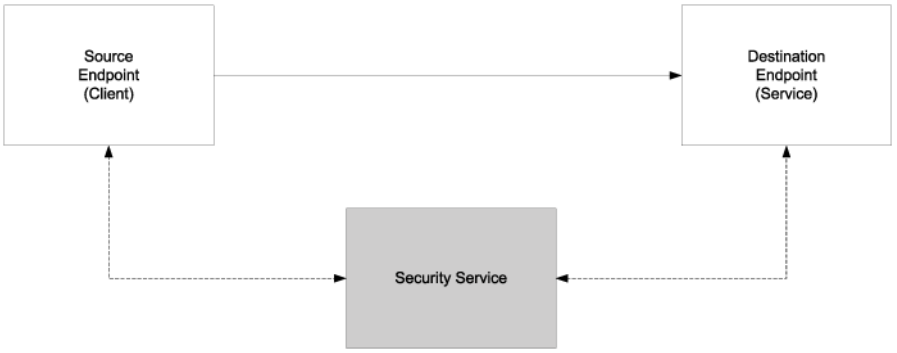


Figure 8.5 Both endpoints invoke the security service out-of-band. As always, security functionality is encapsulated by the security service. Unlike before, both the source and destination endpoints invoke the service, which generates and validates the security tokens.

As in use cases 1 and 2, WS-Addressing and AON are not relevant, as messages are not routed via the security service.

PROS AND CONS

While this use case solves a technical problem, it imposes an additional programming burden on both endpoints. This is a general solution that is suitable for several situations.

All the use cases we outlined so far share an essential characteristic: the security service is invoked out-of-band (that is, the security service lies outside the message path). Let us now consider a radically different possibility. The source endpoint (or a previous intermediary[1] in the message path) can submit the message to a security service rather than the destination endpoint.

8.2.4 Use case 4: Security service as an explicit intermediary

As we described in chapter 3, SOAP allows intermediaries—nodes other than the source and destination endpoints—to see and act on a message. In this use case, we consider implementing a security service as an intermediary in the message path.

[1] Intermediaries in the SOAP message path are discussed in section 3.5.

Figure 8.6 illustrates this idea. The source endpoint (or a previous intermediary in the message path) submits the message to the security service. Upon making the necessary checks, the security service in turn forwards the message to the intended recipient.

Figure 8.6 Security service implemented as an explicitly addressed intermediary in the message path. Unlike in previous cases, no endpoint has to invoke the security service. The message is handed off by the source endpoint to the security service, which in turn relays the message to the destination endpoint, after enforcing security.

As the message is being explicitly routed via the security service by the source endpoint (or a previous intermediary in the message path), we say that the security service is acting as an *explicit* intermediary.

Relevant standards and technologies

Of the standards and technologies described in table 8.2, the following are relevant in this use case.

- *WS-Addressing* As the security service intermediary needs to forward the message to the destination endpoint, the source endpoint needs to communicate the real destination endpoint's address to the security service. This need is fulfilled by WS-Addressing.
- *SAML* The security service can express its findings in the form of standard SAML assertions.

WS-Trust and the SAML protocol are not relevant in this use case, as the source endpoint does not invoke the security service separately; instead, the source endpoint simply routes its messages via the security service. A security service implemented as an intermediary will accept arbitrary messages, as it will have to process messages intended for any destination endpoint.

AON is not relevant here unless the security service is hosted on a network device (as opposed to a server).

Pros and cons

The upside is that the security service gets to see the entirety of the messages and not just the source endpoint's credentials. This allows the security service to offer more functionality than was possible in use cases 1, 2, and 3. For example, the security service can scan the messages to look for attacks that target common vulnerabilities in destination endpoints. We will discuss some of the common vulnerabilties in web services in chapter 10.

The downside is that every source endpoint will have to explicitly route messages through the security service. That means increased complexity and dependence on the details of the security service invocation. At the destination endpoint, the situation is not bad unless we want to have two-way messaging. In request-response type scenarios that require both requests as well as responses to be processed by the security service, both endpoints need to be aware of the security service and route messages through it.

There is one additional complexity. As the security service intermediary needs to forward the message to the destination endpoint, the source endpoint needs to communicate the real destination endpoint's address to the security service. WSAddressing, introduced in chapter 3, can be used for this purpose.

8.2.5 Use case 5: Security service as an implicit intermediary

Our issue with explicit routing is increased programming complexity. If we transparently route the messages via a security service, we get all the benefits of explicit routing without programming it in each of the endpoints. It looks like figure 8.7.

Transparent routing is not possible without support from infrastructure such as network devices. If we can program the network devices to force the traffic through a security service, we can make security decisions based on the messages.

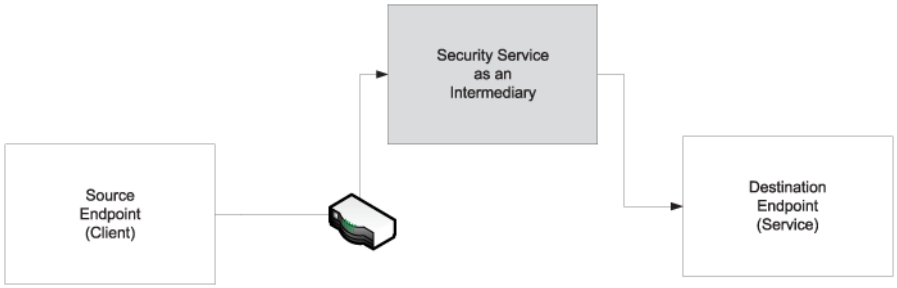


Figure 8.7 Security service implemented as an intermediary in the message path with the help of a network device. It is similar to the earlier case depicted in figure 8.6. The difference here is that the addition of the security service to the message path is done by a network device, without any endpoint knowing about it.

RELEVANT STANDARDS AND TECHNOLOGIES

Of the standards and technologies described in table 8.2, the following are relevant in this use case.

- *WS-Addressing* Just as in use case 4, if the security service intermediary is to forward the message to the destination endpoint after it examines a message, it needs to know the real destination endpoint's address. WS-Addressing helps in this.
- *SAML* The security service can express its findings in the form of standard SAML assertions.

- *AON* This technology enables network devices to understand applicationlevel context and make decisions on whether to route an application-level message via the security service. Furthermore, the network can itself act as the security service, as you will see when we describe AON in appendix E.

Just as for the previous use case, WS-Trust and the SAML protocol are not relevant, as the security service is not invoked explicitly.

Pros and cons

The big upside, compared to the previous use case, is that endpoints need not be burdened with the task of routing messages via the security service. The downside is also obvious: We need AON devices in order to implement this use case.

This completes our use case analysis for a security service. At the start of this chapter, when considering the idea of security as a separate service (see section 8.1.1), we identified three main technical questions we need to answer. Here are those three questions once again.

1. Who invokes the security service?
2. How is the security context communicated to the destination endpoint?
3. What is the interface for the security service?

We answered the first question by identifying five possible ways in which a security service may be invoked. Let's now shift our focus to the second and third questions. The next two sections (8.3 and 8.4) will show how to address the second question using SAML, a language that allows a security service to express its findings. In section 8.6, we will address the third question.

8.3 Conveying the findings of a security service: SAML

In chapters 3-7, you have already seen one way of decoupling security logic from business logic, albeit not in the form of a separate security service. In the examples shown in previous chapters, all the logic of security enforcement is owned by JAX-RPC handlers that are separate from the JAX-RPC service endpoints that provide business logic for services. Saving the findings of security handlers in a JAXRPC `MessageContext` is a common technique that we have repeatedly used in these examples.

For instance, `JAASAuthenticationHandler` in example 2 (depicted in figure 4.2) saved the authenticated `Subject` instance in `MessageContext` because, even though we decoupled the security logic from the business logic, the business logic still depends on some of the findings of the security service.

For example, the `BrokerageService` in our examples needed to know the identity of the user who is placing an order. It may not just be the user identity that the business logic depends on. Information on the groups the user belongs to, the user's preferences, and the user's location may also be needed. All of this information may become available to the security service when it authenticates a user.

The entity that manages security—be it a security-related JAX-RPC handler or a security service—needs to communicate some of its findings (collectively referred to as the *security context*) to the service endpoint and any other node in the message path that lies downstream of the security service.

When we decoupled security logic from business logic using a JAX-RPC handler, we could convey the findings of the security handler using `MessageContext`, an in-memory data structure, as the JAX-RPC handler ran in the same process[1] as the service endpoint. The same technique cannot be used when we move the security logic out of the service endpoint process into a separate security service process. We need an alternative technique that does not rely on inmemory structures.

SAML fulfills this need by providing a language for expressing the findings (or assertions) of the security service. In this section, we will introduce SAML and its usage in web services security. In particular, we will describe:

- The structure of a SAML assertion
- Three standard types of statements you can make within a SAML assertion
- The techniques used to protect SAML assertions from forgery, tampering, and replay

In section 8.5, we will also show you the code for a sample security service that uses OpenSAML, an open source library for producing SAML assertions.

Let us first look at the basics of SAML assertions.

8.3.1 SAML assertion basics

As the name suggests, SAML provides a markup language for representing security assertions. These assertions are created by the entity responsible for security enforcement (such as a security service) to convey its findings to other entities that depend on those findings. In this section, we will show you the structure of a SAML `Assertion` element.

Listing 8.1 is an example that shows the structure of a SAML `Assertion` element.

Listing 8.1 A sample SAML assertion

```
<Assertion xmlns="urn:oasis:names:tc:SAML:1.0:assertion"
  MajorVersion="1" MinorVersion="1"
  AssertionID="MySAMLAssertion1"
  Issuer="http://manning.com/xmlns/samples/soasecimpl/cop"
  IssueInstant="2005-09-19T18:07:08.419Z">
  <Conditions
    NotBefore="2005-09-19T18:07:08.419Z"
    NotOnOrAfter="2005-09-19T18:12:08.419Z"/>
  ...
</Assertion>
```

Indicates SAML version

1 Identifier for this assertion

2 URI identifying assertion issuer

3 Time at which this assertion was issued

Conditions under which this assertion is valid 4

5 One or more statements (not shown here)

[1] By *process*, we mean a running instance of a program. Different parts of the same process can exchange data by simply writing to and reading from the same in-memory location. Data exchange between different processes on the same machine or different machines is lot more complicated and requires a data exchange standard.

A SAML assertion must always have an identifier ❶ and must always say who is making the assertion ❷ and when the assertion is made ❸. A SAML assertion may also explicitly state the `Conditions` ❹ under which it is valid. For instance, in the previous listing, the assertion specifies the time period during which it is valid.

The assertion in listing 8.1 did not say much. That is because we skipped the statements ❺ that make up the meat of the `Assertion`. SAML defines three standard statements you can make within an `Assertion`:

- Authentication statement
- Attribute statement
- Authorization decision statement

We will describe each of these in the following sections, with the help of examples.

8.3.2 AuthenticationStatement: Asserting authentication results

An example of a simple assertion could be user authentication. After a security service authenticates a user, if it wants to provide that information to endpoints (and any downstream node in the message path), it can use this assertion. Listing 8.2 shows how an assertion can say who was authenticated, how, and when.

Listing 8.2 A sample SAML assertion making an authentication statement

```
<Assertion ...>
...

<AuthenticationStatement

  AuthenticationInstant="2005-09-19T18:07:08.379Z"
AuthenticationMethod="...:SAML:1.0:am:password">

  <Subject>
    <NameIdentifier
      NameQualifier="manning.com"
      Format="...:SAML:1.1:nameid-format:emailAddress">
      chap@manning.com
    </NameIdentifier>
</Subject>
</AuthenticationStatement>

...
</Assertion>
```

Statement about authentication done by issuer

Time when authentication took place

Mechanism used for authentication

Information on the authenticated subject

In this listing, the issuer of the assertion is stating that a user identified by the email address `chap@manning.com` has been authenticated using a password-based authentication scheme. We are skipping detailed descriptions of `AuthenticationStatement` (and other statements we discuss in this section), as you can easily decipher the details from the listing. You can refer to the SAML specification (cited in the "Suggestions for further reading" section at the end of this chapter) for complete descriptions.

Consider the example of a security service. When invoked, implicitly or explicitly, it can validate the username and password. When it does, it can use SAML to assert the identity of the user. It can assert even more information, such as the groups the user belongs to, the user's preferences, and the user's location. All this information is part of the security context for a user, which is needed by services to authorize users and customize information for them.

Let us next look at how the security service can assert user's attributes. :

8.3.3 AttributeStatement: Asserting user attributes

As we mentioned, we need to assert various attributes about the user to reduce the burden of security information on the consumer. An endpoint can use such information to make access control decisions or simply customize its behavior. Listing 8.3 shows how an assertion can state attributes of a subject, in this case, the groups the user belongs to:

Listing 8.3 A sample SAML assertion making an attribute statement

```
<Assertion ...>
...
<AttributeStatement>
  <Subject>
    <NameIdentifier
      NameQualifier="manning.com"
      Format="...:SAML:1.1:nameid-format:emailAddress">
      chap@manning.com
    </NameIdentifier>
</Subject>
  <Attribute
    AttributeName="memberOf"
    AttributeNamespace="http://manning.com/saml/attrns">

    <AttributeValue>authors</AttributeValue>
    <AttributeValue>soasecimpl</AttributeValue>

  </Attribute>
</AttributeStatement>

...
</Assertion>
```

A statement about the attributes of a subject

Identity of the subject

Namespace qualified name of an attribute

One or more values of the attribute

In this sample, the assertion is stating that the subject, `chap@manning.com`, is a `memberOf` two groups named `authors` and `soasecimpl`. An `AttributeStatement` can provide values for any number of a subject's attributes. We have only showed one in this listing.

In addition to asserting a user's identity and attributes, a security service may also have the responsibility of asserting which actions a user is allowed to carry out and which he isn't. Next, let us see how SAML makes that possible.

8.3.4 AuthorizationDecisionStatement: Asserting authorization decisions

The security service may convey the kind of access granted to the subject for various resources using `AuthorizationDecisionStatements`. Listing 8.4 shows an example.

Listing 8.4 A sample SAML assertion making an authorization decision statement

```
<Assertion ...>
  ...

  <AuthorizationDecisionStatement
    Resource="http://manning.com/ebooks/soasecimpl/"
    Decision="Permit">

    <Subject>
      <NameIdentifier
        NameQualifier="manning.com"
        Format="...:SAML:1.1:nameid-format:emailAddress">
        chap@manning.com
      </NameIdentifier>
    </Subject>

  <Action Namespace="http://manning.com/saml/actionns">
      Annotate
    </Action>
  ...
  </AuthorizationDecisionStatement>

  ...
</Assertion>
```

Statement about an authorization decision

1 Identity of the access-controlled resource

2 The authorization decision

3 Identity of the subject

4 Action for which the decision is given

The assertion in this example states that the user, `chap@manning.com` 3, is permitted 2 to `Annotate` 4 the resource identified by the URI, `http://manning.com/ebooks/soasecimpl/` 1.

An `AuthorizationDecisionStatement` can record the authorization `Decision` for more than one `Action` on a `Resource`. We have only showed one in this listing.

As you can see from these examples, SAML allows a security service to communicate its findings using three kinds of statements.

1. Authentication statements to indicate that the identity of the caller has been verified by the security service.
2. Attribute statements to indicate the caller's attributes, such as the list of groups/roles the caller belongs to.
3. Authorization decision statements to indicate the actions the caller is allowed to carry out on one or more resources.

At the start of this section, we explained how the decoupling of security logic into a separate security service necessitates a mechanism such as SAML to communicate the findings of the security service to the service endpoint and other nodes in the SOAP message path. There is one more repercussion of moving security logic out into a sep-

arate security service. All communication between the security service and the service endpoint now needs to be secured just like any other data on the wire. In other words, SAML assertions are as vulnerable to forgery, tampering, and replay attacks as any other data on the wire. Appendix D describes in detail various techniques you can use to secure a SAML assertion against these vulnerabilities.

There's much more to SAML than what we covered here. An important aspect in the context of a security service is the SAML protocol that specifies an interface for explicitly invoking a security service. We will discuss it in section 8.5.2.

SAML answers a very important question for implementing security as a service: How does the security service communicate its findings (and do so securely) to the service endpoint? You now understand the answer to this question. Let us solidify that understanding by looking at the implementation for a sample security service that uses SAML. Once we do that, we will return to address other challenges in the implementation of a shared security service:.

8.4 Example implementation using OpenSAML

In this chapter, we are discussing the idea of offering security as a service. So far, you have seen the different use cases for a security service and how the findings of a security service can be represented using SAML.

We are now in a position to implement one of the use cases identified in section 8.2. WS-Addressing, described in chapter 3, and SAML, described in the previous section, are all you need to implement use case 4. In this section, we will show you a sample implementation of this use case.

In the sample implementation shown here, we will create and parse WSAddressing–defined XML elements using W3C DOM/SAAJ APIs—just as we created and parsed, for example, `UsernameToken` in chapter 3. On the other hand, when it comes to creating and parsing SAML-defined elements, we will use a higher-level API provided by OpenSAML, like we used the Apache XML Security library in chapters 6 and 7 to create and parse elements defined by the XML Encryption and XML Signature standards.

Figure 8.8 shows the components involved in the example and the data exchanges between them. The source endpoint explicitly routes the request to a security service that authenticates requests based on WS-Security `UsernameToken`. If the authentication is successful, the security service adds its findings as SAML assertions and forwards the message to the destination endpoint that is identified by the WS-Addressing headers in the message. We will use the brokerage service you have seen in the examples throughout this book as the destination endpoint.

Even though this example will only illustrate a security service that authenticates usernames and passwords, you can easily extend it using the code shown in previous three chapters to use other authentication schemes, encryption, and signatures.
Table 8.3 provides the instructions to set up and run the example.[1]

[1] One or more known issues in Apache Axis 1.x prevent this example from running successfully. See appendix A for a description of these issues.

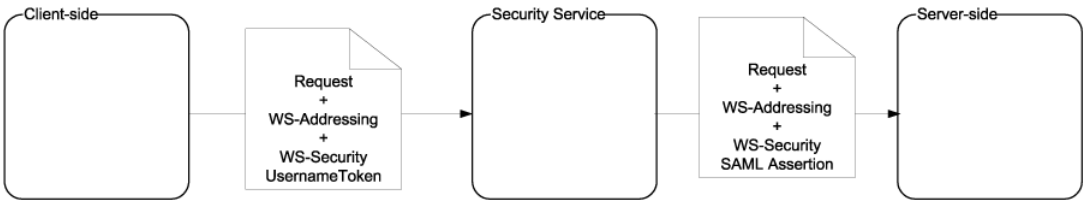


Figure 8.8 Overview of the example used to illustrate the concept of a centralized security service. The message is routed from the client-side to the server-side via the security service. Notice that the username and password token is replaced by the SAML assertion as the message is processed by the security service.

Table 8.3 Steps to run example 6, which illustrates a shared security service

Step	Action	How to
1	Set up your environment.	If you have not already set up the environment required to run the examples in this book, please refer to chapter 2 to do so. `ant deploy` installs all the examples.
2	Set up JAAS configuration.	As the security service will be performing authentication via JAAS, customize the JAAS configuration file, set up the JAVA_OPTS environment variable, and restart Tomcat as described by steps 2, 3, and 4 (respectively) in table 4.1.
3	If it is not already running, start TCP monitor.	Run `ant tcpmon` so that you can observe the conversation. Check the "XML Format" check box to allow `tcpmon` to format shown requests and responses.
4	Run the example.	Run `ant demo -Dexample.id=6`. You should be able to view the request-response messages as they are captured in the `tcpmon` console.

You should now see the execution of web service calls as captured by tcpmon. You will see two requests, one to `/axis/services/proxy` and one to `/axis/services/example6`. The first of these is the request from the client endpoint to the security service, and the second is the request forwarded by security service to the brokerage service.

We will walk through the code that implements the client-side, the security service, and the server side to help you see how all the components come together. Let's tackle the client-side code first.

8.4.1 Client-side implementation

Take a look at figure 8.8 to figure out what we need to do on the client-side. You will recognize that you already know how to do half the job—adding `UsernameToken` to the request. You have seen the code for this chapter 4. The other half of the job is to add the WS-Addressing headers to the request and route the request via the security service. Here we will describe the code for this other half of the functionality.

Figure 8.9 zooms into the full details of the client-side implementation.

In addition to the `ClientSideWSSecurityHandler` that you saw in chapter 4, you can see that we use an additional handler named `AddressingHandler` in this example. There's a little more going on here than just the addition of this new handler. There are two additional tasks we are carrying out in this code compared to what you saw in chapter 4. These tasks are:

1. Routing the request to the security service rather than the service endpoint.
2. Preserving the endpoint address using a WS-Addressing header entry. Let's dive into the implementation details for each of these tasks separately.

ROUTING A REQUEST VIA THE SECURITY SERVICE

There is no standard way to reroute a request in a JAX-RPC client. In chapter 2, we saw that JAX-RPC provides a client three different ways of invoking a web service. A client can use a pregenerated stub, a dynamic proxy, or the JAX-RPC dynamic invocation interface. In all three cases, JAX-RPC does not provide a standard way to explicitly route a request via an intermediary. In the example shown in figure 8.8, we

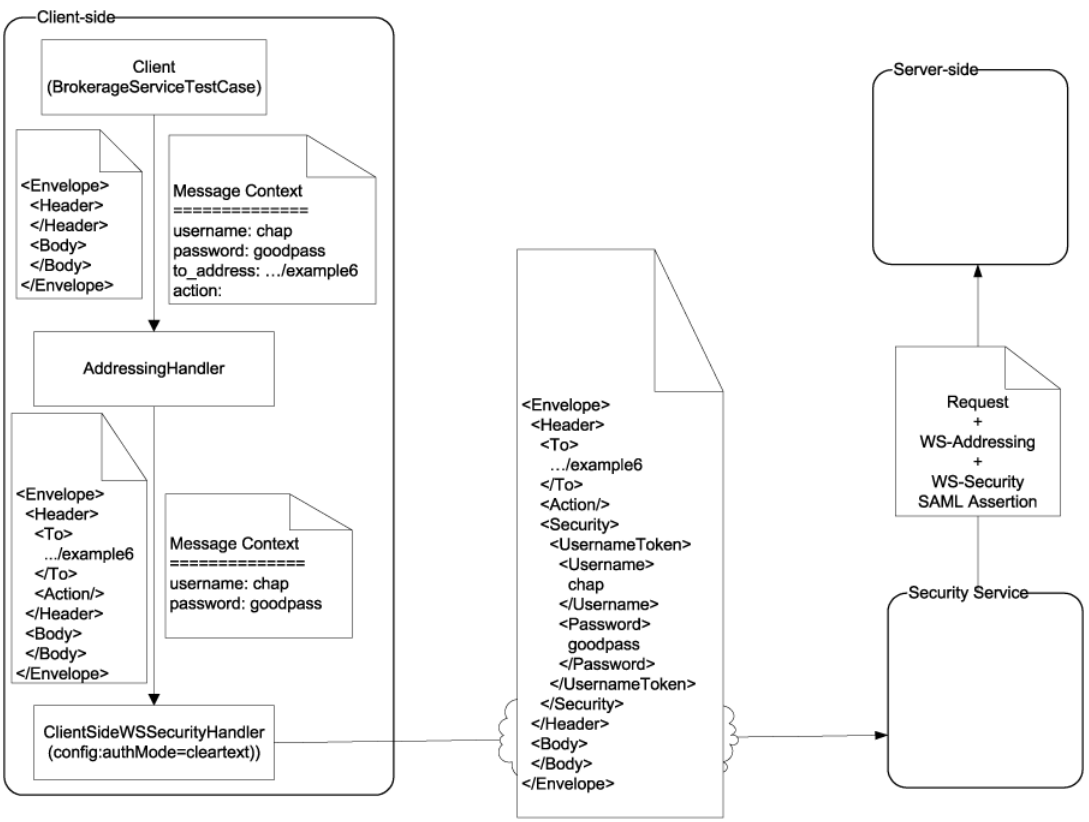


Figure 8.9 Details of client-side implementation in example 6. The client side adds two pieces of information using handlers: the address and the username and password.

pregenerated a stub using Apache Axis's `WSDL2Java` tool and replaced the endpoint address in one of the generated files, `BrokerageServiceLocator.java`, to trick Axis into explicitly routing the request to the security service instead of the target endpoint. This kind of a hack may or may not be possible in other web services engines. In such cases, you will have to configure your network to route the request implicitly to the security service.

ADDING A WS-ADDRESSING HEADER ENTRY

Preserving the endpoint address using a WS-Addressing header is accomplished using a JAX-RPC handler named `AddressingHandler`. As shown in figure 8.9, the client sets up the destination endpoint address and SOAP Action URI values in message context. The `AddressingHandler` reads these values from the message context and creates WS-Addressing headers, `wsa:To` and `wsa:Action`. The code required to do this is similar to the code you saw in chapter 3 for `ClientSideWSSecurityHandler`. Instead of username and password, we have the destination endpoint address and SOAP action URI; and instead of a `Security` header with a `UsernameToken` element in it, we have `To` and `Action` headers to create. Given that you have seen this pattern before, we will skip line-by-line explanation of `AddressingHandler` here. See `example6/AddressingHandler` in the example code base if you wish to review the code.

That's all there really is to the client-side implementation. Let us now take a look at the code in the sample security service.

8.4.2 Security service implementation

The functionality of the security service in this example can be divided into three parts:

1. Authenticating the request by verifying the username/password provided by the client.
2. Creating a SAML authentication statement and adding it to the WS-Security header.
3. Forwarding the message to the endpoint.

You are already familiar with the code needed to accomplish the first of these three parts—`WSSecurityUsernameHandler` and `JAASAuthenticationHandler` described in chapter 4. What you have yet to see is the code for the second and third parts, which we will describe here.

Figure 8.10 zooms into the full implementation details of the sample security service.

In addition to the `WSSecurityUsernameHandler` you have already seen in chapter 4, observe that we have two new components—`SAMLCreationHandler` and `ProxyService`—that you have not seen before. These two components are responsible for creating SAML assertions and forwarding the request to the service endpoint. We will describe the code that goes into each of these components next. Let's start with the code in the `SAMLCreationHandler`.

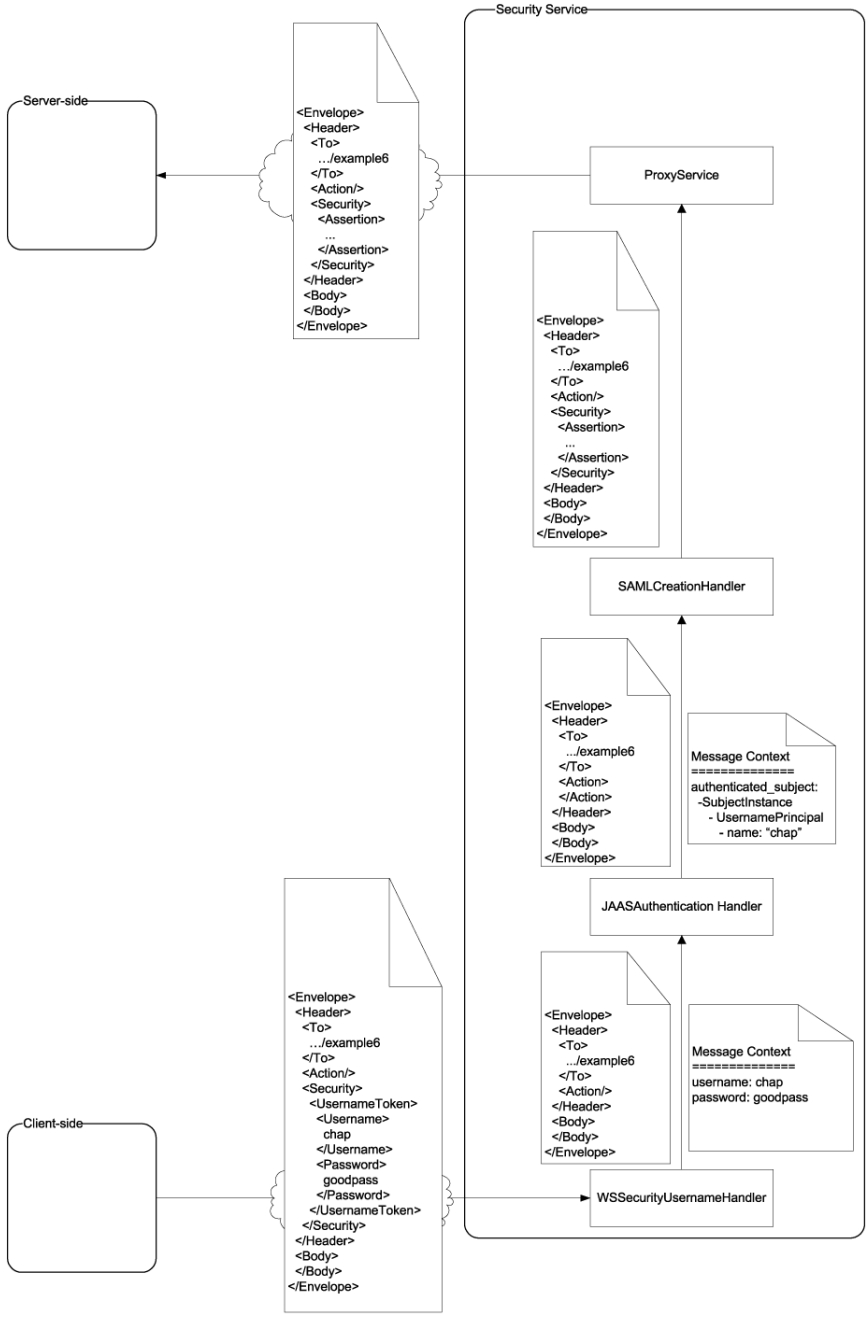


Figure 8.10 Details of the security service implementation. The security service has a JAASAuthenticationHandler to validate the username and password and a SAMLCreationHandler to add the SAML assertion to the message. The proxy service is used to forward the message to the server side.

CREATING SAML STATEMENTS AND ASSERTIONS: SAMLCREATIONHANDLER

Code in a `SAMLCreationHandler` is very much like the code in client-side JAX-RPC handlers such as `ClientSideWSSecurityHandler`, which you have seen before. The authenticated subject information placed in the message context by `JAASAuthenticationHandler` is used by the `SAMLCreationHandler` to create a SAML assertion with an `AuthenticationStatement` in it. Instead of creating a SAML assertion and its contents using low-level DOM APIs, we will use OpenSAML, an open-source library for creating and consuming SAML.

Listing 8.5 Code snippet from `SAMLCreationHandler`

```
String username = (String) messageContext.getProperty
    (Constants.USERNAME_MSG_CONTEXT_PROPERTY);
if (username == null) {
    throw new RuntimeException
    ("Username cannot be null if authentication succeeded");
}
```

Looks up authenticated username

```
String authenticationMethod = (String)
    messageContext.getProperty
    (Constants.AUTHENTICATION_METHOD_MSG_CONTEXT_PROPERTY);
```

Looks up authentication method

```
SAMLSubject samlSubject = new SAMLSubject();
samlSubject.setNameIdentifier
    (new SAMLNameIdentifier
            (username,
            null, //optional name qualifier - can be used to
                  //indicate the domain username belongs to
            null //optional format URI to indicate name
                 //format. e.g., is the name a email
                 //address? or a X.509 subject name? or
                 //or a windows domain name?
            ));
```

Creates a SAML subject

```
SAMLAuthenticationStatement authStmt =
    new SAMLAuthenticationStatement();
authStmt.setSubject(samlSubject);
Calendar instant = Calendar.getInstance();
authStmt.setAuthInstant(instant.getTime());
authStmt.setAuthMethod(authenticationMethod);
```

Creates an AuthenticationStatement

```
SAMLAssertion samlAssertion = new SAMLAssertion();
samlAssertion.addStatement(authStmt);
samlAssertion.setIssuer(faultActor);
```

Creates a SAML assertion

```
samlAssertion.setIssueInstant(instant.getTime());
samlAssertion.setNotBefore(instant.getTime());
instant.add(Calendar.SECOND,validityInterval);
samlAssertion.setNotOnOrAfter(instant.getTime());
```

Sets validity interval for the assertion

```
SOAPHeaderElement wsaToElement =
    Utils.getHeaderByNameAndActor
        (soapEnvelope,
         Constants.WS_ADDRESSING_TO_QNAME,
         null, //no specific roles
         true); //use headers targeted at ultimate dest
if (wsaToElement == null) {
    throw new RuntimeException("To Address not found");
}
String toAddress = wsaToElement.getValue();
```

Looks up To address from WS-Addressing header

```
SAMLAudienceRestrictionCondition audienceCondition =
    new SAMLAudienceRestrictionCondition();
audienceCondition.addAudience(toAddress);
samlAssertion.addCondition(audienceCondition);
```

Adds AudienceRestriction-Condition to assertion

```
securityElement.appendChild
    (soapPart.importNode(samlAssertion.toDOM(), true));
```

Adds the assertion to WS-Security header

The net effect of this code is to produce a SAML assertion element with an `AuthenticationStatement` (like in listing 8.2), a validation period (like in listing 8.1), and an `AudienceRestrictionCondition` (like in listing 8.5).

This completes the description of the code in `SAMLCreationHandler`. We will not show the code for securing the assertion using encryption and signatures, as you have already seen in chapters 6 and 7 (respectively) example code for implementing encryption and signatures.

Referring back to figure 8.10, there is only one component in the example security service that you have yet to understand: `ProxyService`. Let's look at that next.

FORWARDING THE MESSAGE TO THE ENDPOINT: PROXYSERVICE

The last component of the security service needs to forward the message to the next hop along the message path. In this example, the destination endpoint is the next hop. As we implemented the security service using JAX-RPC handlers in Axis, the last component is a web service implemented in Axis. We have named the web service `ProxyService`, as it simply forwards requests to the destination endpoint and routes responses back to the client.

The `ProxyService` is quite different from the `BrokerageService` you have seen in all the examples until now. We implemented `BrokerageService` as an RPC-style service in Axis. We cannot do the same for `ProxyService`, as the service does not explicitly provide operations that a client can invoke. In `ProxyService`, we simply forward the entirety of the SOAP request message to the destination endpoint and return the resulting response message to the client. We implement `ProxyService` as a *message-style* service in Axis. A message-style service deals directly with the request and response SOAP messages instead of relying on Axis to parse SOAP messages into Java objects on the way in and serialize Java objects into SOAP messages on the way out.

In the rest of this section, we will show you how to implement `ProxyService` as a message-style service in Axis. We will first explain how you can declare a message-style service in Axis. We will then show you the code that goes into `ProxyService`.

To declare `ProxyService` as a message-style service in Axis, we need to first cre-ate a WSDD file by hand. We did not do this for `BrokerageService` because its WSDD was generated by the `WSDL2java` tool. If we are deploying the `BrokerageService` and `ProxyService` in the same Axis instance, we can simply edit the WSDD file generated for the former and add the following service description to it; otherwise, we can clone the WSDD file generated for the former and replace the service description as shown in listing 8.6.

Listing 8.6 Deployment descriptor for the `ProxyService`

```
<service name="proxy" style="message">
  <handlerInfoChain>
    <handlerInfo classname="...example6.WSSecurityUsernameHandler">
        <parameter name="usernameTokenMandatory" value="false"/>
    </handlerInfo>
    <handlerInfo classname="...example6.JAASAuthenticationHandler">
        <parameter name="jaasAppName" value="soasecimpl"/>
    </handlerInfo>
    <handlerInfo classname="...example6.SAMLCreationHandler"/>
    <role soapActorName=".../soasecimpl/cop"/>
  </handlerInfoChain>
  <parameter name="className" value="...example6.ProxyService"/>
  <parameter name="allowedMethods" value="relayInAxis"/>
</service>
```

Now that we know how to declare `ProxyService` as a message-style service in Axis, let's shift our focus to the code in `ProxyService`. Axis requires methods that provide a message-style service to have one of the signatures shown in listing 8.7.

Listing 8.7 Four possible signatures for methods that implement a message-style service in Axis

```
public Element [] method(Element [] bodies);
public SOAPBodyElement [] method (SOAPBodyElement [] bodies);
public Document method(Document body);
public void method(SOAPEnvelope req, SOAPEnvelope resp);
```

In our declaration of `ProxyService` (see listing 8.6), we only named one method for `ProxyService`, and that is `relayInAxis`. None of the four method signatures allowed by Axis are ideal for this method, as all we want to do is forward the request message to the destination endpoint and return the response we receive from the endpoint to the caller of the `ProxyService`. We do not act on the message in any particular way.

The fourth signature is the one that comes closest to our needs. Even though it provides access to the request and response envelopes, it is not an ideal fit for our needs for the following two reasons.

1. There can be lot more in a SOAP message than just a SOAP envelope. You saw in chapter 7 that a SOAP message can have any number of attachments.
2. Depending on the transport used, there may be special transport-level headers that need to be set. In our example, HTTP is the transport and we need to set an HTTP header named `SOAPAction` for the destination endpoint to correctly process the request.

What we really need is the ability to access the request `SOAPMessage` instance and set the response `SOAPMessage` instance as opposed to just getting access to request and response `SOAPEnvelope` instances. This is what we ended up doing for implementing the `relayInAxis` method:

- Adopt the fourth signature shown in listing 8.7 but do not rely on the request and response envelopes provided as arguments.
- Get access to the request message from the message context (the JAX-RPC `SOAPMessageContext` API provides this facility).
- Relay the request message to the service endpoint using the JAXM API.
- Take the response message returned by the JAXM call and relay it back to the client endpoint by setting it as the response message in `ProxyService`. The JAX-RPC `SOAPMessageContext` API does not provide a method for setting the response message. We use a Axis-specific `MessageContext` method call in this step.

To separate out the axis-specific and portable code, most of the logic in these steps is implemented in a different method named `relay`. The `relayInAxis` method simply invokes the `relay` method and takes care of Axis-specific logic. Both the methods are shown in listing 8.8.

Listing 8.8 Code snippet from the `ProxyService` implementation

```
public void relayInAxis
  (SOAPEnvelope requestEnv, SOAPEnvelope responseEnv) {

  SOAPMessage requestMessage = messageContext.getMessage();
  SOAPMessage relayResponseMessage = relay(requestMessage);

  ((org.apache.axis.MessageContext)messageContext).
    setResponseMessage
      ((org.apache.axis.Message)relayResponseMessage);
}

public SOAPMessage relay(SOAPMessage relayMessage) {
    logger.debug("received a request to relay");

    try {
        //look up the destination address and SOAPAction
        //from the WS-Addressing headers in the request

        //... this portion of code not shown ...
```

```
            //=== now, call the target service ===

            SOAPConnection connToDestination =
                SOAPConnectionFactory.newInstance().
                createConnection();
            URLEndpoint destinationEndpoint =
                new URLEndpoint(toAddress);
            relayMessage.getMimeHeaders().setHeader
                (Constants.HTTP_SOAP_ACTION_HEADER, action);

            return connToDestination.call
                (relayMessage, destinationEndpoint);
        } catch (Exception e) {
            throw createSOAPFault(e);
        }
    }
```

How did the `ProxyService` know where to forward the request message? The destination endpoint information is available in the WS-Addressing headers created by the `AddressingHandler` on the client-side. The code required to extract the destination address and action from WS-Addressing headers is quite similar to the code you have seen in chapters 3-7 to extract security tokens from WS-Security headers. We will skip reviewing that portion of the code here.

This discussion completes the security service implementation. Let us move on to discuss the code on the server-side.

8.4.3 Server-side implementation

Figure 8.11 zooms into the server-side implementation.

The server-side implementation in this example needs a handler that can consume the SAML `AuthenticationStatement` provided by the security service and set the username in message context for the benefit of `BrokerageService`.

The security service in this example does not encrypt/sign SAML assertions, but as that is to be expected in general, we extend the `ServerSideWSSecurityHandler` you saw in chapter 7 and add to it the code for processing a SAML assertion. The resulting handler code invokes the `processSAMLAssertion` method shown in listing 8.9 whenever it encounters a SAML assertion in the WS-Security header.

Listing 8.9 Code to process a SAML assertion with an authentication statement and set the authenticated subject information in message context

```
private void processSAMLAssertion
    (Element samlAssertionElement,
    SOAPMessageContext soapContext, String faultActor) {
    logger.debug("Processing SAML Assertion");
    try {
        SAMLAssertion samlAssertion =
            new SAMLAssertion(samlAssertionElement);
        Iterator samlStatementsIter =
            samlAssertion.getStatements();
        while(samlStatementsIter.hasNext()){
```

```
                Object stmt = samlStatementsIter.next();
                if (stmt instanceof SAMLAuthenticationStatement) {
                    SAMLAuthenticationStatement authStmt =
                        (SAMLAuthenticationStatement) stmt;
                    soapContext.setProperty
                        (Constants.USERNAME_MSG_CONTEXT_PROPERTY,
                         authStmt.getSubject().
                         getNameIdentifier().getName());
                    break;
                }
            }
        } catch (Exception e) {
            createFaultInContextAndThrow
            (soapContext,
             Constants.SOAP_SERVER_FAULT_CODE,
             "Error processing SAML: ",
             faultActor, e);
        }
    }
```

This code is self-explanatory so we will not explain it line-by-line. Observe that this code sets the authenticated subject information (username in this example) in the message context, just as the `JAASAthenticationHandler` did in chapter 4. This is so

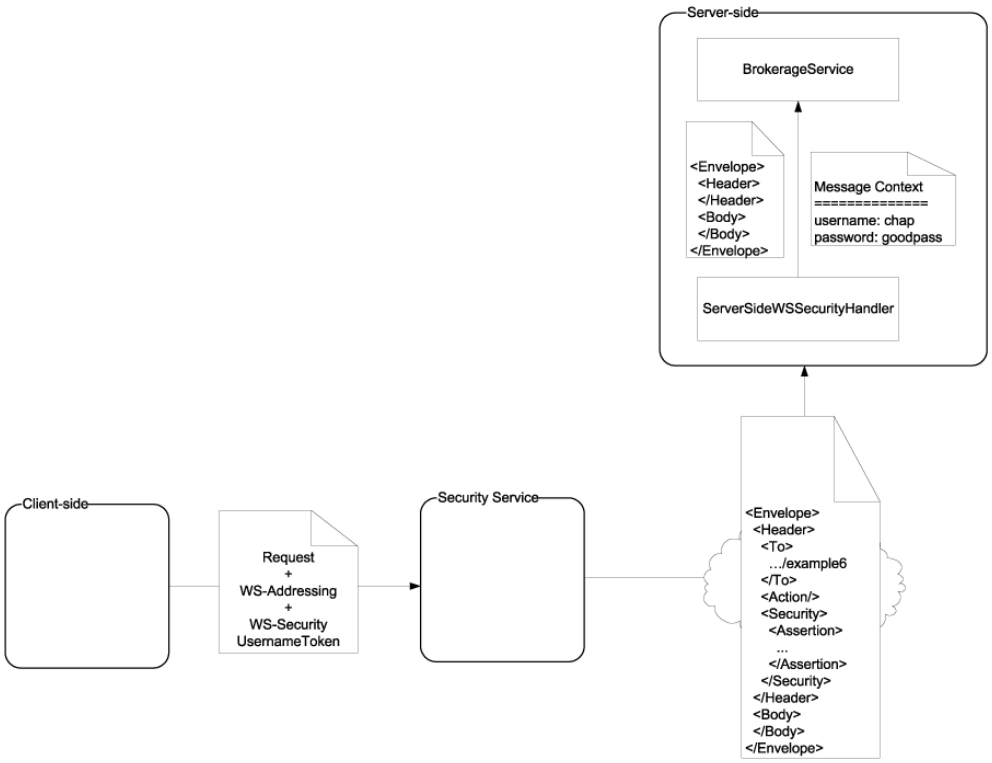


Figure 8.11 Details of server-side implementation. The server-side handler needs to understand and validate the assertion, and finally extract the username and password from the message.

that the `BrokerageService` can use the authenticated user's identity in its business logic. The `BrokerageService` code remains unchanged from what you have seen in previous examples. So, we will not show it here.

This completes our example to illustrate the use of SAML in a security service. At the start of this chapter, when we introduced the idea of a shared security service, three important questions came up:

1. Who invokes the security service?
2. How is security context communicated to the destination endpoint?
3. What is the interface for the security service?

We answered the first of these questions in section 8.2 by identifying five different use cases for a security service. We introduced SAML to answer the second question. We will proceed to the third question next.

8.5 Standards for security service interfaces

In section 8.2, we discussed different possibilities for how to invoke of a security service. In three out of the five use cases we discussed, the security service is explicitly invoked by the sender, the receiver, or both. What interface should a security service provide for such explicit invocations? Is there a standard interface security service consumers can rely on?

The answer depends on the functionality you want from the security service. WS-Trust and the SAML protocol (not the same as SAML assertions) are two standards that specify an explicitly invokable interface for a security service. In this section, we will introduce you to both of these standards. But before we do that, we should tell you what to expect and what not to expect from these standards (see callout).

NOTE ***What to expect and what not to expect from an interface for a security service***
In this chapter, you have already seen an example of a security service. We showed you a sample security service that functions as an intermediary to authenticate SOAP requests. We also mentioned that our sample security service can be easily extended to provide additional functionality such as encryption/decryption and creation/verification of digital signatures. Knowing this, you may now have certain expectations about the functionality a security service can provide. Note that we have only shown you so far what a security service can do as an intermediary. Not all functionality that a security service can offer as an intermediary may make sense for a security service that is invoked explicitly. For example, while authentication and authorization are often delegated to a security service using explicit invocation, the same cannot be said of encryption/decryption and creation/verification of signatures. Retransmitting all or substantial amounts of large messages to a security service in order to encrypt/decrypt/sign/verify signatures is, in general, not considered scalable.

This understanding of the difference in scope between a security service that is offered as an intermediary and one that is offered for explicit invocation will help you appreciate why the interfaces standardized by WS-Trust and the SAML protocol cater to certain kinds of security needs and not to other needs.

We will now briefly introduce you to the interfaces standardized by WS-Trust and the SAML protocol. Let's start with WS-Trust.

8.5.1 WS-Trust

WS-Trust[1] describes interfaces for a security service that can issue, validate, renew, and cancel security tokens. In our case, the security tokens we are dealing with are SAML assertions. As you recall, SAML provides only the language to express the security information. Without help of a standard such as WS-Trust, we cannot issue, validate, renew, or cancel any security token.

WS-Trust describes a service called a *security token service (STS)* that issues security tokens. The interface for the STS is designed to meet the requirements of a wide variety of situations. Here, we describe four commonly encountered scenarios:

1. *The client requests a token to access a web service from STS* A web service client can request that STS issue a security token that can be used to access a web service. When making the request to STS, the client can use WSSecurity or transport-layer mechanisms to identify itself. The security token issued by STS in this case can serve as proof of identity, a confirmation of privilege, or even as a key that can be used to encrypt and/or sign messages. For example, the returned token can be a SAML assertion or a Kerberos ticket or a key.
2. *The intermediary invokes STS to do authentication/authorization* An intermediary such as a web services gateway can intercept requests and invoke STS to check the identity claims made by the client and, optionally, authorize them. If the client claims are authenticated, STS can issue a security token (such as a SAML assertion) to convey its findings to the intermediary that invoked it or to the destination web service.
3. *The service endpoint invokes STS to do authentication/authorization* A web service endpoint can itself refer a caller's security claims to the STS and obtain a statement of findings as a security token.
4. *The intermediary/target service invokes STS to exchange one security token for another* This is a combination of scenario 1 with scenario 2 or scenario 3. A web service client can submit a security token provided by one STS to an intermediary/target service that in turn consults a different STS. The net effect is that one security token is exchanged for another token that the end service understands and accepts.

As you can see, STS needs a generic interface that can serve different types of security tokens in different kinds of situations. STS's designers met this challenge by coming up with a generic request-response protocol. In this section, we will describe this protocol first at a high level. Subsequently, we will explain the structure of the request and response messages with examples. The examples will be restricted to illustrating the issu-

[1] WS-Trust is expected to become more popular as it forms the basis for CardSpace (formerly "InfoCard"), Microsoft's replacement for Passport.

ance of a new security token. Once you follow these examples, you can look up the WS-Trust specification and easily understand the interfaces for token validation, renewal, and cancellation, which are needed for security service to be invoked by other services.

Let's get started with a high-level overview of the STS interface standardized by WS-Trust.

STS INTERFACE

As we described before, WS-Trust defines a standard interface for security services that issue security tokens. Any security service that supports this standard interface is referred to as an STS. Figure 8.12 illustrates the interaction between an STS and a party that seeks to procure a security token from the STS.

Figure 8.12 Security token service interface defined by WS-Trust: a request for a security token (RST) is met with an RST response (RSTR).

The party that seeks to obtain a security token from the STS may be an endpoint (client/server) or an intermediary. In all cases, the party requesting the security token sends a SOAP message containing a `RequestSecurityToken` (RST) elementto the STS. If the STS is satisfied that the caller qualifies for the security token it is requesting, the STS then responds with a SOAP message containing an RSTR element; if not, it responds with a SOAP fault.

So, the actual details of the STS interface are all in the RST and RSTR elements. We will describe them next, starting with RST first.

REQUESTSECURITYTOKEN (RST)

Before taking a deep dive into the makeup of a RST element, let's look at the big picture. A party requesting a security token from the STS needs to do things in its request:

1. Describe the kind of security token it is requesting. For example, the requestor needs to say whether it wants a Kerberos ticket or a SAML assertion.
2. Prove that it is qualified to get the security token it is requesting. This may simply mean authenticating with the STS using valid credentials.

The first is what an RST element facilitates. The second can be accomplished via WS-Security, using the authentication techniques you learned in chapter 4 (password-based auth), chapter 5 (Kerberos), and chapters 6 and 7 (PKI-based auth). Listing 8.10 emphasizes this division of responsibilities between the RST element defined by WS-Trust and the `Security` header entry defined by WS-Security.

Listing 8.10 Structure of a request to an STS

```
<soapenv:Envelope ...>
  <soapenv:Header>
    <wsse:Security soapenv:actor="...">
      ...
    </wsse:Security>
  </soapenv:Header>

  <soapenv:Body>
    <wst:RequestSecurityToken xmlns:wst="...">
      ...
    </wst:RequestSecurityToken>
  </soapenv:Body>
</soapenv:Envelope>
```

WS-Security header with caller's claims

RST describing the requested security token

Now that you understand the context in which an RST element is used, let's look deeper into the makeup of an RST. Listing 8.11 shows an example RST element.

Listing 8.11 Example of an RST element

```
<wst:RequestSecurityToken xmlns:wst="...">
  <wst:TokenType>
    .../wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1
  </wst:TokenType>

  <wst:RequestType>
    http://schemas.xmlsoap.org/ws/2005/02/trust/Issue
  </wst:RequestType>

  <wsp:AppliesTo xmlns:wsp="...">
    <wsa:EndpointReference xmlns:wsa="..."
      xmlns:impl=".../samples/soasecimpl">
      <wsa:Address>
        .../axis/services/example6
      </wsa:Address>
      <wsa:PortType>
        impl:Brokerage
      </wsa:PortType>
      <wsa:ServiceName>
        impl:BrokerageService
      </wsa:ServiceName>
    </wsa:EndpointReference>
  </wsp:AppliesTo>

  <wst:Claims Dialect="...">...</wst:Claims>

  <wst:Entropy>
    ...
  </wst:Entropy>

  <wst:Lifetime>
    <wsu:Created>2006-03-16T02:17:41</wsu:Created>
    <wsu:Expires>2006-03-16T02:22:41</wsu:Expires>
  </wst:Lifetime>
</wst:RequestSecurityToken>
```

1 Type of requested security token

2 Indicates request goal

3 Service the caller wishes to invoke using issued token

4 Info on claims needed in the requested security token

5 An optional cryptographic key

6 Suggested validity period for the returned key

Let's walk through the content of the RST element in this example to further understand what goes into an RST.

The `TokenType` element ❶ indicates the type of security token the caller desires from the STS. Token types are identified using URIs laid down in WS-Security Token Profiles (standards describing the use of a token in WS-Security). In this example, the caller is asking STS to return a SAML 1.1 assertion using the URI reserved by WS-Security's SAML Token Profile.

The `RequestType` element ❷ indicates what the request is for: token issuance, validation, renewal, or cancellation. In this example, the caller is requesting that STS issue a new token by using the URI shown as the `RequestType`.

The `AppliesTo` element ❸ provides a reference to the service the caller wishes to contact using the issued security token. This allows the STS to use its knowledge of the target service provider's policies and determine the type of token to issue. The `TokenType` element ❶ we seen earlier is optional if the `AppliesTo` is specified, and vice versa. Both can be specified as well, but in that case, STS is free to override the `TokenType` requested, as it probably has better knowledge of service policies than the caller. We should point out here the use of elements defined by WS-PolicyAttachment and WS-Addressing, as indicated by the namespace prefixes `wsp` and `wsa`, respectively. The `AppliesTo` element is defined by WS-PolicyAttachment. We will discuss policy-related standards in the next chapter. The `EndpointReference` element is defined by WS-Addressing as a standard mechanism for referring to a service endpoint. In this example, we are relying on the endpoint address, port type, and service name defined in the service WSDL to create a reference to the service.

The caller asked for a SAML 1.1 assertion in this example, but it didn't indicate the kinds of statements it wants the STS to make, did it? Should the STS issue an `AuthorizationStatement` along with an `AuthenticationStatement`? Should it add an `AttributeStatement` as well? The STS might infer the answers to these questions based on its knowledge of the service provider's policies (assuming the target service is identified using the `AppliesTo` element, as in this example). Or, the caller can explicitly request from the STS specific kinds of statements using the `Claims` element ❹. WS-Trust relies on WS-SecurityPolicy to provide syntax for specifying the needed claims. We will discuss WS-SecurityPolicy in the next chapter.

The optional `Entropy` element ❺ allows the caller to provide a cryptographic key to the STS. We will describe the motivation behind this when describing the STS response.

The optional `Lifetime` element ❻ can be used by the caller to suggest a length of time for which the returned security token should be valid. The STS may disregard the timeframe suggested by the caller.

In summary, the RST element helps callers of STS to specify the kind of security token they want, the claims they want in the token, and how long they want to use the returned token. All this information is needed by STS to provide a suitable token for the caller. Let's now take a look at what the STS returns in response to an RST.

REQUESTSECURITYTOKENRESPONSE (RSTR)

You saw in figure 8.12 that an STS returns an RSTR in response to an RST if the caller is found to be eligible for the requested security token. Before we get into the details of what an RSTR contains, it helps to discuss a few important considerations that went into RSTR's design.

Security tokens can have a lot of information in them. Authentication and authorization statements, information on attributes of the caller, dynamically generated cryptography keys, and signatures to guarantee the integrity of all these items may all be part of a security token. Given this wide range of possibilities, nobody other than the STS and the service for which the security token is intended should assume that they can look into the security token and understand it.

At the same time, there are a few things that callers often need to know about the security token issued by STS. For example, the caller may need to know the type of token returned and its expiry time. If the token returned by the STS provides a dynamically generated key that the caller can use to encrypt the service request, the caller will need a copy of the key in order to do the encryption. In addition, the caller will need the ability to tell the service, "I am encrypting my request message using the key provided to me by STS." In other words, the caller needs the ability to refer to the token returned by STS.

STS needs to provide a lot more than just an opaque security token in RSTR. The example response shown in listing 8.12 will help you understand how WSTrust accomplishes this.

Listing 8.12 Example of an RSTR element

```
<wst:RequestSecurityTokenResponse>
  <wst:TokenType>
    .../wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1
  </wst:TokenType>

  <wst:RequestedSecurityToken>
    <!-- an encrypted and signed SAML assertion -->
  </wst:RequestedSecurityToken>

  <wsp:AppliesTo>...</wsp:AppliesTo>

  <wst:RequestedAttachedReference>
    <wsse:SecurityTokenReference
      wsu:Id="MySAMLAssertion1Ref1">
      <wsse:KeyIdentifier
        ValueType="...#SAMLAssertionID">
        MySAMLAssertion1
      </wsse:KeyIdentifier>
    </wsse:SecurityTokenReference>
  </wst:RequestedAttachedReference>
```

1 Issued security token type

2 Issued token, possibly encrypted and signed

3 Reference to service for which the token is issued

4 Reference to use if the issued token is in the same doc

```
  <wst:RequestedUnattachedReference>
    <wsse:SecurityTokenReference
     wsu:Id="MySAMLAssertion1Ref2">
      <saml:AuthorityBinding
        Binding="...:SAML:1.0:bindings:SOAP-binding"
        Location="http://..."
        AuthorityKind="samlp:AssertionIdReference"/>
      <wsse:KeyIdentifier
        ValueType="...#SAMLAssertionID">
        MySAMLAssertion1
      </wsse:KeyIdentifier>
    </wsse:SecurityTokenReference>
  </wst:RequestedUnattachedReference>

  <!--
  <wst:RequestedProofToken>...</wst:RequestedProofToken>
  <wst:Entropy>...</wst:Entropy>
  -->
  <wst:Lifetime>
     <wsu:Created>2006-03-16T02:17:41</wsu:Created>
    <wsu:Expires>2006-03-16T02:22:41</wsu:Expires>
  </wst:Lifetime>
</wst:RequestSecurityTokenResponse>
```

❺ Reference to use if the issued token is not in the same doc

❻ A token caller can use to prove possession of the issued token

❼ A random key generated and used by the STS to create ❻

❽ Validity period for issued token

Just as we did when discussing an RST example, let's now walk through each of the child elements in the example RSTR to further understand what goes into a RSTR element.

The `TokenType` element ❶ indicates the type of token issued. In this example, the token issued is a SAML 1.1 assertion.

The `RequestedSecurityToken` element ❷ contains the issued token, possibly encrypted and signed to secure it from every party other than the service endpoint for which the token is issued.

The `AppliesTo` element ❸ provides a reference to the service endpoint for which the token is issued, in case the caller asked for a service-specific token.

The `RequestedAttachedReference` element ❹ provides a reference to the issued token (SAML assertion in this example) for use in documents carrying the issued token.

The `RequestedUnattachedReference` ❺ element provides a reference to the issued token for use in any document. In this example, the reference provides the identifier of an assertion that can be retrieved from a SAML authority. Attributes on the `SAMLAuthorityBinding` element describe the protocol, location, and query to use when contacting the SAML authority. You will see more details on the SAML protocol in the next section.

The `RequestedProofToken` element ❻ provides a token the caller can use to prove possession of the token issued by the STS. (Note that this and the `Entropy` element are commented out, as they are not needed in this example. We are only showing them here for the sake of completeness.) This element becomes useful when STS wants to

provide the caller and the target service with two different copies of a dynamically generated cryptographic key, each encrypted differently.

You have seen the `Entropy` element ❼ in the RST, and we promised to explain its significance when we got here. Callers who want to ensure that the STS is generating cryptographic keys randomly enough can do so by providing a random key that they themselves generate. When a caller provides STS with entropy, STS has three choices.

1. Adopt the caller provided key as is. In this case, RSTR will not have `RequestedProofToken` and `entropy` elements.
2. Discard the caller-provided key and generate its own. In this case, `RequestedProofToken` is used to convey the generated key.
3. Combine the caller-provided key with another key STS generates itself. In this case, the key generated by STS is communicated using the `entropy` element in RSTR and the computation used to combine the keys is indicated by a `<wst:ComputedKey>AlgorithmURI</wst:ComputedKey>` under `RequestedProofToken`.

The `Lifetime` element ❽ specifies the period of validity for the issued token.

You now understand the STS interface provided by WS-Trust for requesting issue of new tokens. With this understanding, you can easily follow the interfaces described in the WS-Trust specification for validating, renewing, and canceling security tokens. You should certainly read the WS-Trust spec now if you need a complete understanding of the standard for your work. See the "Suggestions for further reading" section at the end of the chapter for a link to the specification.

So far you've seen how WS-Trust can help a security service provide security tokens to callers. If a security service implements WS-Trust, then its callers can request using RST and get a response in RSTR. These interfaces are independent of any security token. For our purpose, the only security tokens we are interested in are SAML assertions. Is there a simpler protocol if we restrict our security tokens only to SAML assertions? It turns out that there is one such protocol, aptly named the SAML protocol, which we will describe next.

8.5.2 SAML protocol

If all we are interested in issuing, validating, renewing, and canceling SAML assertions then SAML protocol is much simpler than WS-Trust. It is often employed in SSO solutions for web applications. Here we will show how it can be used over SOAP.

Like WS-Trust, the SAML protocol proposes a request-response protocol. In this section, we will describe this protocol with the help of simple examples. Let us first take a look at how a request is made in SAML protocol.

MAKING A REQUEST

The structure of a request is simple enough to understand through an example. Here is an example request querying the security service to see if the named subject is permitted to execute the identified resource.

Listing 8.13 Example of a request using SAML protocol over SOAP

```
<soapenv:Envelope ...>
  <soapenv:Header>
    <wsse:Security soapenv:actor="...">
      ...
    </wsse:Security>
  </soapenv:Header>

  <soapenv:Body>
    <samlp:Request ...>
      <samlp:AuthorizationDecisionQuery>
        <saml:Subject>
          <saml:NameIdentifier
            NameQualifier="manning.com"
            Format="...:nameid-format:emailAddress">
            chap@manning.com
          </saml:NameIdentifier>
        </saml:Subject>
      </samlp:AuthorizationDecisionQuery>
    </samlp:Request>
  </soapenv:Body>
</soapenv:Envelope>
```

WS-Security header with caller's security claims

Request element defined by SAML protocol

Authorization query for the named subject

This example is quite straightforward. The caller submits his credentials using the WS-Security header entry and requests that the security service return a SAML assertion with an `AuthorizationDecisionStatement` (see listing 8.4) for the subject identified in the query.

Let's now look at an example response to understand how a security service supporting the SAML protocol can respond to requests.

RECEIVING A RESPONSE

The response, just like the request, constitutes the body of a SOAP message. We will skip showing the SOAP envelope, header, and body elements here.

Listing 8.14 Example of a response from a security service supporting the SAML protocol

```
<samlp:Response>

  <samlp:Status>
    <samlp:StatusCode Value="samlp:Success"/>
  </samlp:Status>
```

Indication of the request's success or failure

```
  <saml:Assertion ...>
    <samlp:AuthorizationDecisionStatement
      Resource=".../axis/services/example6"
      Decision="Permit">

      <saml:Subject>
        <saml:NameIdentifier
          NameQualifier="manning.com"
          Format="...:nameid-format:emailAddress">
          chap@manning.com
        </saml:NameIdentifier>
      </saml:Subject>

      <saml:Action
        Namespace="...:SAML:1.0:action:rwedc">
        Execute
      </saml:Action>

    </samlp:AuthorizationDecisionStatement>
  </saml:Assertion>
</samlp:Response>
```

Returned SAML assertion

As you can see from this example, the security service first states whether the request succeeded or failed, and if it succeeded, returns one or more assertions. For a complete list of status codes, refer to the SAML specification cited in the "Suggestions for further reading" section at the end of this chapter.

Let's recap what you have learned in this section. Implementing security as a service can usher in multiple benefits, but there are a few challenges that need to be tackled to make it possible. One of the key challenges is to define a standard interface that a security service can offer to callers. In this section, we saw two standards, WS-Trust and SAML protocol that address this challenge.

8.6 Summary

Up until this chapter, we focused on addressing one security aspect at a time. We did this deliberately to help you understand each of the fundamental building blocks of SOA security. Now that you understand all the fundamental building blocks, we have shifted our focus in this chapter toward figuring out the best way to assemble those building blocks to provide complete solutions that are maintainable, manageable, scalable, and auditable.

The most obvious way to add security to services and service consumers is to build the security logic into every service and every consumer. Such an approach leads to manageability and maintainability problems. So, we considered a different approach: implementing security as a service.

We then looked at some use cases for a security service and figured we need a few more standards and technologies to be in place if we are succeed in implementing security as a service. In particular we showed the need for:

1. Standards/technologies that enable the use of intermediaries, such as a security service
2. Standard ways to communicate the findings of a security service
3. Standard interfaces for invoking security services

In chapter 3, we introduced WS-Addressing and SOAP processing rules for intermediaries to fulfill the first of these three needs. In this chapter, we introduced SAML for the second and WS-Trust/SAML protocol for the third. We also showed you a working example of a security service intermediary using Apache Axis, OpenSAML, and JAXM. If you followed these examples, you can see how we can implement a security service that can issue SAML assertions that can be used by service consumers and providers. Such a service is manageable, scalable, maintainable, and auditable.

A technology that we have often referred to in this chapter, AON, is described in appendix E. AON technology allows the network to be a provider of security services.

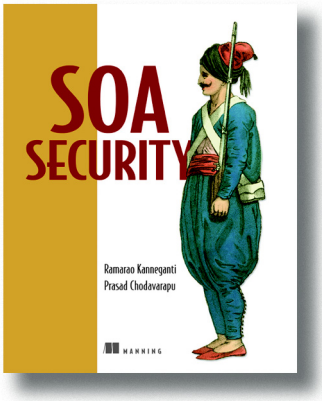
Another appendix that adds to the material in this chapter is appendix D, where we describe how to secure SAML assertions against forgery, tampering, and replay.

While the idea of security as a service is indeed technically feasible as you have seen in this chapter, its adoption in the real world is not often easy. Factoring out all the security logic that has been hard-wired into endpoints is impractical. A hybrid approach is needed to realize the benefits of a security service as much as possible, even while accommodating legacy endpoints that take upon themselves the task of security enforcement. We will discuss this topic further in chapter 10.

Several times in this chapter, we made references to the manageability of security solutions. One approach that promises to greatly enhance manageability of SOA security solutions is declarative security or policy-based security. In the next chapter, we will introduce you to the standards that support such an approach.

8.7 Suggestions for further reading

- The SAML specifications are available at http://www.oasis-open.org/committees/ tc_home.php?wg_abbrev=security. In particular, the SAML 1.1 specification described in this book is available at http://www.oasis-open.org/committees/download.php/3400/oasis-sstc-saml-1.1-pdf-xsd.zip.
- The WS-Security SAML Token Profile 1.1 is available at http://www.oasis-open.org/committees/download.php/16768/wss-v1.1-spec-os-SAMLTokenProfile.pdf.
- The WS-Trust specification was developed by an ad hoc industry group. It is available at http://specs.xmlsoap.org/ws/2005/02/trust/WS-Trust.pdf as an initial public release dated February 2005.
- OpenSAML libraries and documentation are available from http://www. opensaml.org/. The version used in this book is 1.1b.
- Windows CardSpace (formerly "InfoCard"), Microsoft's new digital identity "meta" system based on WS-Trust, WS-SecurityPolicy, and WS-MetadataExchange, is described at http://download.microsoft.com/download/6/c/3/6c3c2ba2-e5f0-4fe3be7f-c5dcb86af6de/infocard-guide-beta2-published.pdf.



Anyone seeking to implement SOA Security is forced to dig through a maze of inter-dependent specifications and API docs that assume a lot of prior security knowledge on the part of readers. Getting started on a project is proving to be a huge challenge to practitioners. *SOA Security* seeks to change that. It provides a bottom-up understanding of security techniques appropriate for use in SOA without assuming any prior familiarity with security topics.

Unlike most other books about SOA that merely describe the standards, this book helps readers learn through action, by walking them through sample code that illustrates how real life problems can be solved using the techniques and best practices described in the standards. It simplifies things: where standards usually discuss many possible variations of each security technique, this book focuses on the 20% of variations that are used 80% of the time. This keeps the material covered useful for all readers except the most advanced.

What's inside

- Why SOA Security is different from ordinary computer security, with real life examples from popular domains such as finance, logistics, and Government
- How things work with open source tools and code examples as well as proprietary tools.
- How to implement and architect security in enterprises that use SOA. Covers WS-Security, XML Encryption, XML Signatures, and SAML.

index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

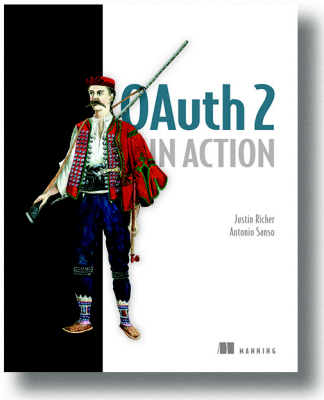
U

V

W

X

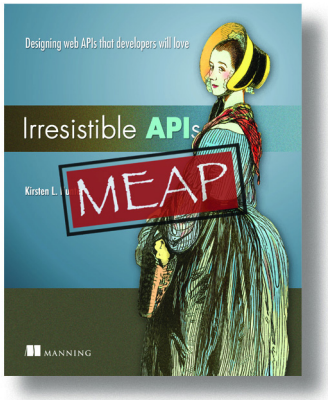
Y



OAuth 2 in Action

by Justin Richer and Antonio Sanso

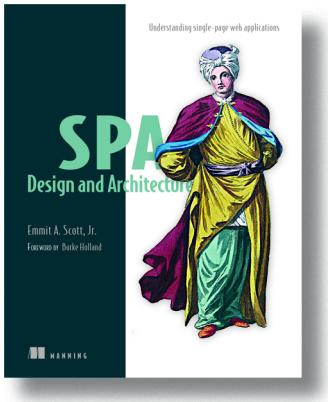
ISBN: 9781617293276
375 pages
$49.99
August 2016



Irresistible APIs: Designing web APIs that developers will love

by Kirsten L. Hunter

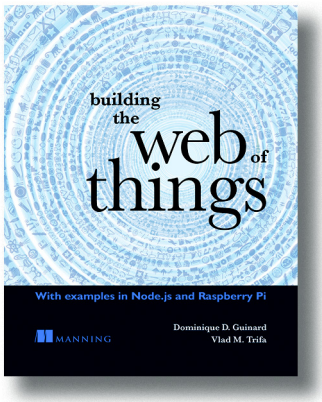
ISBN: 9781617292552
270 pages
$44.99
August 2016



SPA Design and Architecture: Understanding single-page web applications

by Emmit A. Scott, Jr.

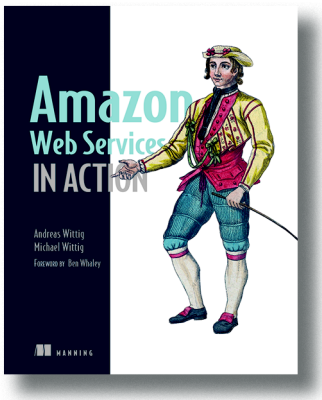
ISBN: 9781617292439
312 pages
$49.99
November 2015



Building the Web of Things

by Dominique D. Guinard and Vlad M. Trifa

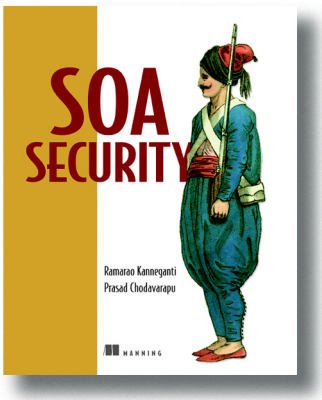
ISBN: 9781617292682
375 pages
$34.99
June 2016



Amazon Web Services in Action

by Michael Wittig and Andreas Wittig

ISBN: 9781617292880
424 pages
$49.99
September 2015



SOA Security

by Ramarao Kanneganti and Prasad A. Chodavarapu

ISBN: 9781932394689
512 pages
$59.99
December 2007