This improved program works exactly the same as the previous one, except the session is saved in Redis. Open the Redis CLI and type this command to get all available keys:

```
[15:09:48] naren:~ $ redis-cli
127.0.0.1:6379> KEYS *
1) "session_VPJ54LWRE4DNTYCLEJWAUN5SDLVW6LN6MLB26W2OB4JDT26CR2GA"
127.0.0.1:6379>
```

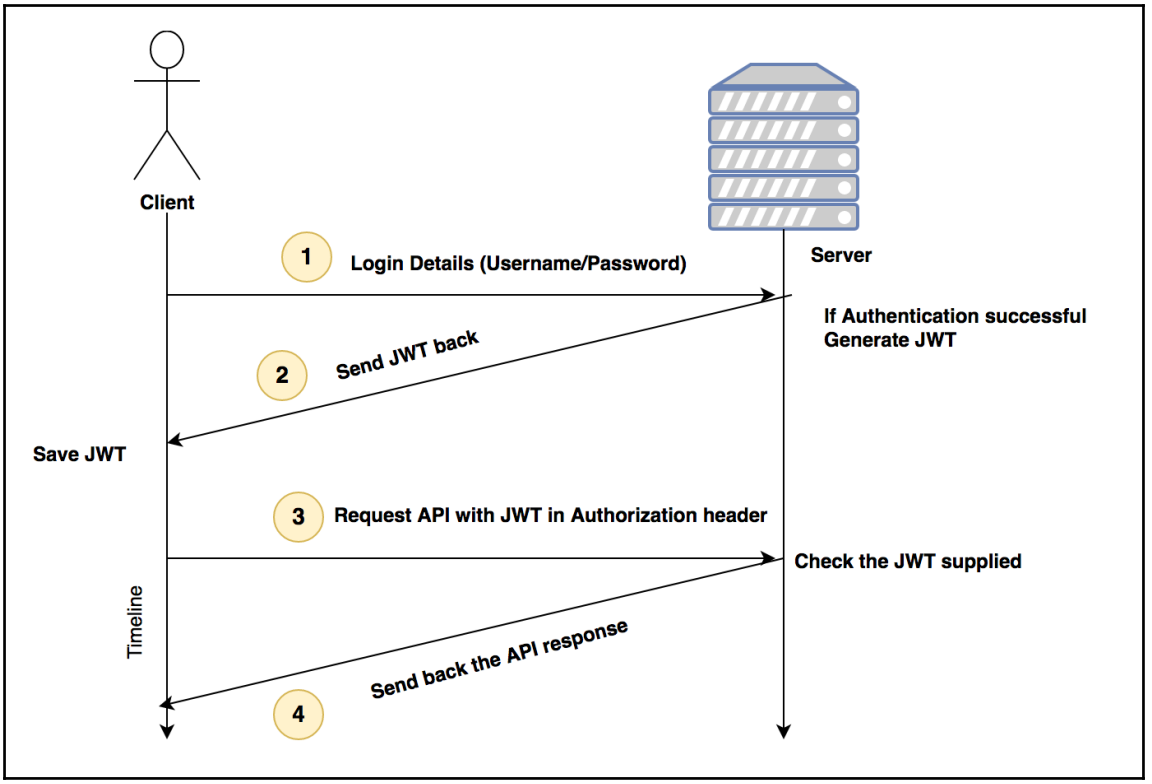
That lengthy `"session_VPJ54LWRE4DNTYCLEJWAUN5SDLVW6LN6MLB26W2OB4JDT26CR2GA"` is the key stored by the `redistore`. If we delete that key, the client will automatically be forbidden from accessing resources. Now, stop the running program and restart it. You will see the session is not lost. In this way, we can save the client session. We can also persist sessions on the SQLite database. Many third-party packages are written to make that much easier.

Redis can serve the purpose of caching for your web applications. It can store temporary data such as sessions, frequently requested user content, and so on. It is usually compared to **memcached**.

Introduction to JSON Web Tokens (JWT) and OAuth2

The previous style of authentication is a plain username/password and session-based. It has a limitation of managing sessions by saving them in the program memory or Redis/SQLite3. The modern REST API implements token-based authentication. Here, tokens can be any strings generated by the server, which allows the client to access resources by showing the token. Here, the token is computed in such a way that the client and the server only know how to encode/decode the token. **JWT** tries to solve this problem by enabling us to create tokens that we can pass around.

Whenever a client passes the authentication details to the server, the server generates a token and passes it back to the client. The client saves that in some kind of storage, such as a database or local storage (in case of browser). The client uses that token to ask for resources from any API defined by the server:



The steps can be summarized more briefly as follows:

1. The client passes the username/password in a `POST` request to the login API.
2. The server authenticates the details and if successful, it generates a JWT and returns it back instead of creating a cookie. It is the client's responsibility to store this token.
3. Now, the client has the JWT. It needs to add this in subsequent REST API calls such as `GET`, `POST`, `PUT`, and `DELETE` in the request headers.
4. Once again, the server checks the JWT and if it is successfully decoded, the server sends the data back by looking at the username supplied as part of the token.

JWT ensures that the data is sent from the correct client. The technique for creating a token takes cares of that logic. JWT leverages the secret key-based encryption.

JSON web token format

All we discussed in the preceding section was circling around a JWT token. We are going to see here what it really looks like and how it is produced. JWT is a string that is generated after performing few a steps. They are as follows:

1. Create a JWT header by doing **Base64Url** encoding on the header JSON.
2. Create a JWT payload by doing **Base64Url** encoding on the payload JSON.
3. Create a signature by encrypting the appended header and payload using a secret key.
4. JWT string can be obtained by appending the header, payload, and signature.

A header is a simple JSON object. It looks like the following code snippet in Go:

```
`{
  "alg": "HS256",
  "typ": "JWT"
}`
```

`"alg"` is a short form for the algorithm (HMAC with SHA-256) used for creating a signature. The message type is `"JWT"`. This will be common for all the headers. The algorithm may change depending on the system.

A payload looks like this:

```
`{
  "sub": "1234567890",
  "username": "Indiana Jones",
  "admin": true
}`
```

Keys in payload object are called claims. A claim is a key that specifies some special meaning to the server. There are three types of claims:

- Public claims
- Private claims (more important)
- Reserved claims

Reserved claims

Reserved claims are the ones defined by the JWT standard. They are:

- iat: issued at the time
- iss: issuer name
- sub: subject text
- aud: audience name
- exp: expiration time

For example, the server, while generating a token, can set an `exp` claim in the payload. The client then uses that token to access API resources. The server validates the token each time. When the expiration time is passed, the server will no longer validate the token. The client needs to generate a new token by logging in again.

Private claims

Private claims are the names used to identify one token from another. It can be used for authorization. Authorization is a process of identifying which client made the request. Multi-tenancy is having multiple clients in a system. The server can set a private claim called `username` on the payload of the token. Next time, the server can read this payload back and get the username, and then use that username to authorize and customize the API response.

`"username": "Indiana Jones"` is the private claim on the preceding sample payload. **Public claims** are the ones similar to private claims, but they should be registered with the IANA JSON Web Token Registry to make it as a standard. We limit the use of these.

A signature can be created by performing this (this is not code, just an illustration):

```
signature = HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

It is simply performing an encryption algorithm on the Base64URL encoded header and payload with a secret. This secret can be any string. It is exactly similar to the secret we used in the previous cookie session. This secret is usually saved in the environment variable and loaded into the program.

Now we append the encoded header, encoded payload, and signature to get our token string:

```
tokenString = base64UrlEncode(header) + "." + base64UrlEncode(payload) +
"." + signature
```

This is how a JWT token is generated. Are we going to do all this stuff manually in Go? No. In Go, or any other programming language, a few packages are available to wrap this manual creation of a token and verification. Go has a wonderful, popular package called `jwt-go`. We are going to create a project in the next section that uses `jwt-go` to sign a JWT and also validate them. One can install the package using the following command:

```
go get github.com/dgrijalva/jwt-go
```

This is the official GitHub page for the project: `https://github.com/dgrijalva/jwt-go`. The package provides a few functions that allow us to create tokens. There are many other packages with different additional features. You can see all available packages and features supported at `https://jwt.io/#libraries-io`.

Creating a JWT in Go

The `jwt-go` package has a function called `NewWithClaims` that takes two arguments:

1. Signing method such as HMAC256, RSA, and so on
2. Claims map

For example, it looks like the following code snippet:

```
token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{
    "username": "admin",
    "iat":time.Now().Unix(),
})
```

`jwt.SigningMethodHS256` is an encryption algorithm that is available within the package. The second argument is a map with claims such as private (here username) and reserved (issued at). Now we can generate a `tokenString` using the `SignedString` function on a token:

```
tokenString, err := token.SignedString("my_secret_key")
```

This `tokenString` then should be passed back to the client.

Reading a JWT in Go

`jwt-go` also gives us the API to parse a given JWT string. The `Parse` function takes a string and key function as arguments. The `key` function is a custom function that validates whether the algorithm is proper or not. Let us say this is a sample token string generated by the preceding encoding:

```
tokenString =
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWluIiwiaWF0IjoiM
TUwODc0MTU5MTQ2NiJ9.5m6KkuQFCgyaGS_xcVy4xWakwDgtAG3ILGGTBgYVBmE"
```

We can parse and get back the original JSON using:

```
token, err := jwt.Parse(tokenString, func(token *jwt.Token) (interface{},
error) {
    // key function
    if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
        return nil, fmt.Errorf("Unexpected signing method: %v",
token.Header["alg"])
    }
    return "my_secret_key", nil
})

if claims, ok := token.Claims.(jwt.MapClaims); ok && token.Valid {
    // Use claims for authorization if token is valid
    fmt.Println(claims["username"], claims["iat"])
} else {
    fmt.Println(err)
}
```

`token.Claims` is implemented by a map called `MapClaims`. We can get the original JSON key-value pairs from that map.

OAuth 2 architecture and basics

OAuth 2 is an authentication framework that is used to create authentication pattern between different systems. In this, the client, instead of making a request to the resource server, makes an initial request for some entity called resource owner. This resource owner gives back the authentication grant for the client (if credentials are successful). The client now sends this authentication grant to another entity called an authentication server. This authentication server takes the grant and returns an access token. This token is the key thing for a client to access API resources. It needs to make an API request to the resource server with this access token and the response is served. In this entire flow, the second part can be done using JWT. Before that, let us learn the difference between authentication and authorization.

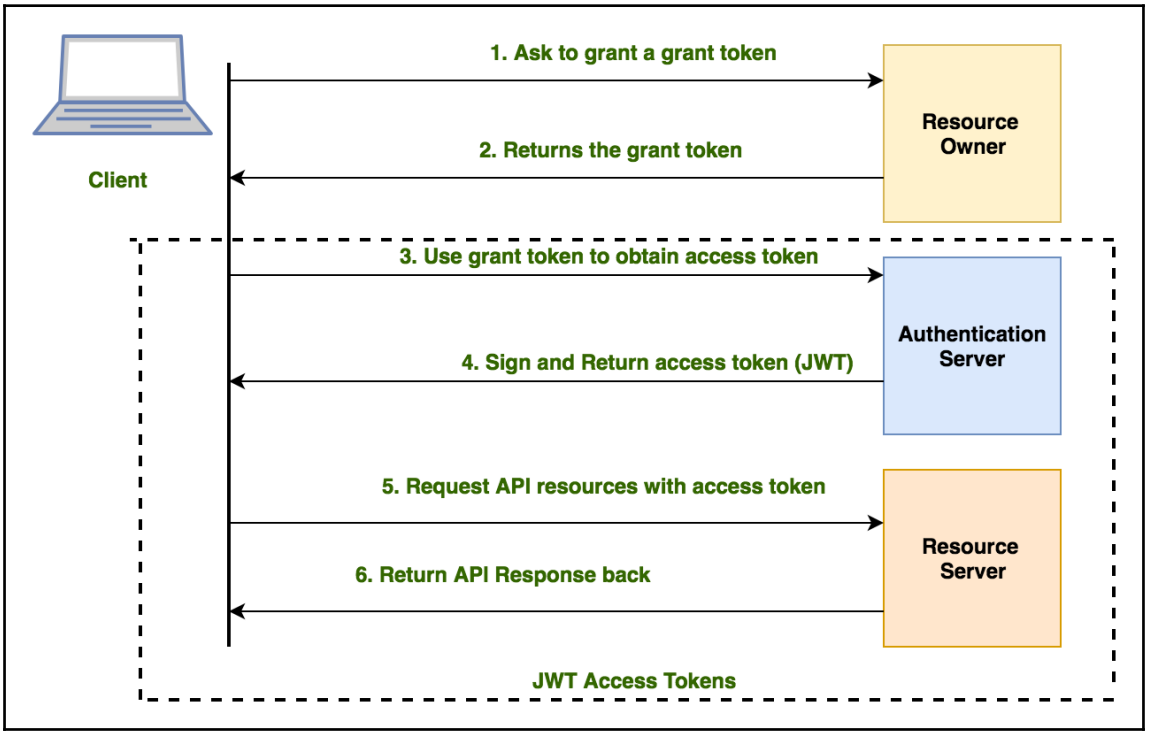
Authentication versus authorization

Authentication is the process of identifying whether a client is genuine or not. When a server authenticates a client, it checks the username/password pair and creates session cookie/JWT.

Authorization is the process of differentiating one client from another after a successful authentication. In cloud services, the resources requested by a client need to be served by checking that the resources belong to that client but not the other client. The permissions and access to resources vary for different clients. For example, the admin has the highest privileges of resources. A normal user's access is limited.

OAuth2 is a protocol for authenticating multiple clients to a service, whereas the JWT is a token format. We need to encode/decode JWT tokens to implement the second stage (dashed lines in the following screenshot) of OAuth 2.

Take a look at the following diagram:



In this diagram, we can implement the dashed section using JWT. Authentication is happening at the authentication server level and authorization happens at the resource server level.

In the next section, let us write a program that does two things:

1. Authenticates the client and returns a JWT string.
2. Authorizes client API requests by validating JWT.

Create a directory called `jwtauth` and add `main.go`:

```
package main
import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "os"
    "time"
    jwt "github.com/dgrijalva/jwt-go"
    "github.com/dgrijalva/jwt-go/request"
    "github.com/gorilla/mux"
)
var secretKey = []byte(os.Getenv("SESSION_SECRET"))
var users = map[string]string{"naren": "passme", "admin": "password"}
// Response is a representation of JSON response for JWT
type Response struct {
    Token string `json:"token"`
    Status string `json:"status"`
}
// HealthcheckHandler returns the date and time
func HealthcheckHandler(w http.ResponseWriter, r *http.Request) {
    tokenString, err :=
request.HeaderExtractor{"access_token"}.ExtractToken(r)
    token, err := jwt.Parse(tokenString, func(token *jwt.Token)
(interface{}, error) {
        // Don't forget to validate the alg is what you expect:
        if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
            return nil, fmt.Errorf("Unexpected signing method: %v",
token.Header["alg"])
        }
        // hmacSampleSecret is a []byte containing your secret, e.g.
[]byte("my_secret_key")
        return secretKey, nil
    })
    if err != nil {
        w.WriteHeader(http.StatusForbidden)
        w.Write([]byte("Access Denied; Please check the access token"))
        return
    }
    if claims, ok := token.Claims.(jwt.MapClaims); ok && token.Valid {
        // If token is valid
        response := make(map[string]string)
        // response["user"] = claims["username"]
        response["time"] = time.Now().String()
        response["user"] = claims["username"].(string)
```

```
        responseJSON, _ := json.Marshal(response)
        w.Write(responseJSON)
    } else {
        w.WriteHeader(http.StatusForbidden)
        w.Write([]byte(err.Error()))
    }
}
// LoginHandler validates the user credentials
func getTokenHandler(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()
    if err != nil {
        http.Error(w, "Please pass the data as URL form encoded",
http.StatusBadRequest)
        return
    }
    username := r.PostForm.Get("username")
    password := r.PostForm.Get("password")
    if originalPassword, ok := users[username]; ok {
        if password == originalPassword {
            // Create a claims map
            claims := jwt.MapClaims{
                "username": username,
                "ExpiresAt": 15000,
                "IssuedAt": time.Now().Unix(),
            }
            token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
            tokenString, err := token.SignedString(secretKey)
            if err != nil {
                w.WriteHeader(http.StatusBadGateway)
                w.Write([]byte(err.Error()))
            }
            response := Response{Token: tokenString, Status: "success"}
            responseJSON, _ := json.Marshal(response)
            w.WriteHeader(http.StatusOK)
            w.Header().Set("Content-Type", "application/json")
            w.Write(responseJSON)
        } else {
            http.Error(w, "Invalid Credentials", http.StatusUnauthorized)
            return
        }
    } else {
        http.Error(w, "User is not found", http.StatusNotFound)
        return
    }
}
func main() {
    r := mux.NewRouter()
    r.HandleFunc("/getToken", getTokenHandler)
```

```
    r.HandleFunc("/healthcheck", HealthcheckHandler)
    http.Handle("/", r)
    srv := &http.Server{
        Handler: r,
        Addr: "127.0.0.1:8000",
        // Good practice: enforce timeouts for servers you create!
        WriteTimeout: 15 * time.Second,
        ReadTimeout: 15 * time.Second,
    }
    log.Fatal(srv.ListenAndServe())
}
```

This is a very lengthy program to digest. First, we are importing `jwt-go` and its subpackage called `request`. We are creating a REST API for two endpoints; one for getting the access token by providing authentication details, and another one for fetching the health check API that authorizes the user.

In the `getTokenHandler` handler function, we are comparing the username and password with our custom defined user map. This can be a database too. If authentication is successful, we are generating a JWT string and sending it back to the client.

In `HealthcheckHandler`, we are taking the access token from a header called `access_token` and validating it by parsing the JWT string. Who is writing the logic of validating? The JWT package itself. When a new JWT string is created it should have a claim called `ExpiresAt`. Refer to the following code snippet:

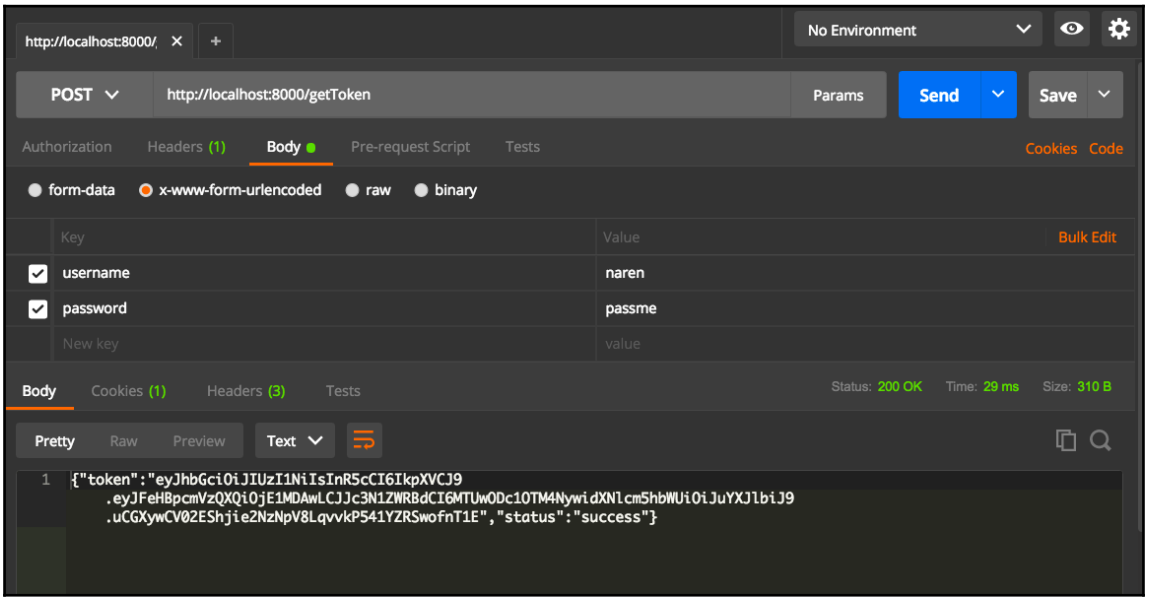
```
claims := jwt.MapClaims{
  "username": username,
  "ExpiresAt": 15000,
  "IssuedAt": time.Now().Unix(),
}
```

The program's internal validation logic looks at the `IssuedAt` and `ExpiresAt` claims and tries to compute and see whether the given token is expired or not. If it is fresh, then it means the token is validated.

Now, when a token is valid, we can read the payload in the `HealthCheckHandler` where we parse the `access_token` string that passed as part of the HTTP request headers. `username` is a custom private claim we inserted for authorization. Therefore, we know who is actually sending this request. For each and every request there is no need for the session to be passed. Each API call is independent and token based. Information is encoded in a token itself.

`token.Claims.(jwt.MapClaims)` returns a map whose values are interfaces, not strings. In order to convert the value to a string, we should do `claims["username"].(string)`.

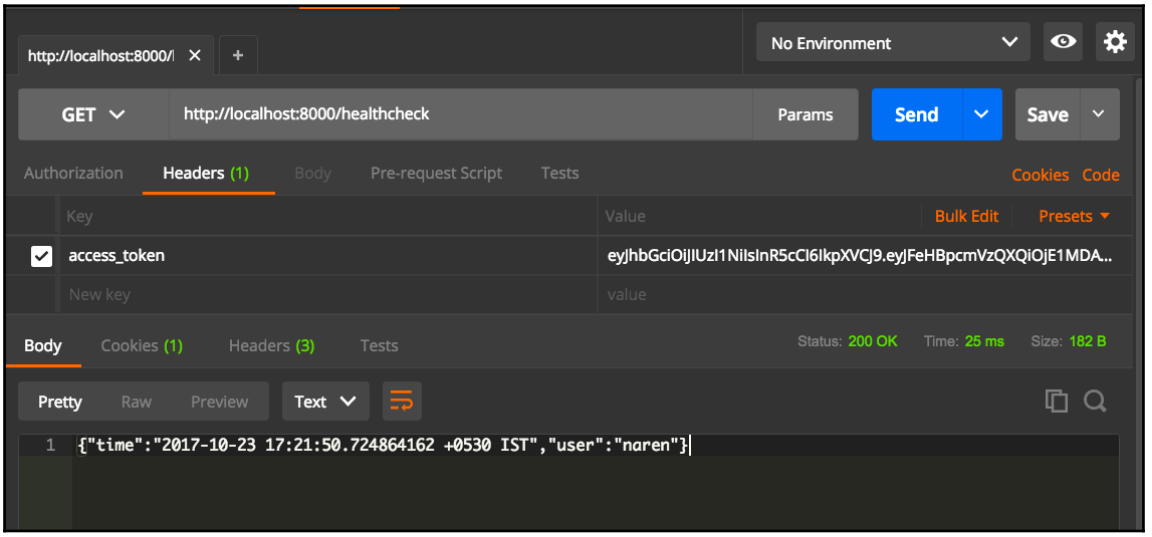
Let us see how this program runs by making requests through the Postman tool:



This returns a JSON string that has a JWT token. Copy it to the clipboard. If you try to make a request to the health check API without passing that JWT token as one of the headers, you will receive this error message instead of JSON :

```
Access Denied; Please check the access token
```

Now, copy that token back and make a `GET` request, adding an `access_token` header with a token string as the value. In Postman, the headers section is available where we can add headers and key-value pairs. Refer to the following screenshot:



It returns the time properly as part of the API response. We can also see which user's JWT token this is. This confirms the authorization part of our REST API. Instead of having the token validation logic in each and every API handler, we can have it as a middleware and make it applicable to all handlers. Refer to `Chapter 3`, *Working with Middleware and RPC*, and modify the preceding program to have a middleware that validates the JWT token.

Token-based authentication doesn't usually provide a log out API or API for deleting the tokens that are provided in session based authentication. The server gives the authorized resources to the client as long as JWT is not expired. Once it expires, the client needs to refresh the token—that is to say, ask the server for a new token.