Migration Guide from v2 -> v3

Version 3 adds several new, frequently requested features. To do so, it introduces a few breaking changes. We've worked to keep these as minimal as possible. This guide explains the breaking changes and how you can quickly update your code.

`Token.Claims` is now an interface type

The most requested feature from the 2.0 verison of this library was the ability to provide a custom type to the JSON parser for claims. This was implemented by introducing a new interface, `Claims`, to replace `map[string]interface{}`. We also included two concrete implementations of `Claims`: `MapClaims` and `StandardClaims`.

`MapClaims` is an alias for `map[string]interface{}` with built in validation behavior. It is the default claims type when using `Parse`. The usage is unchanged except you must type cast the claims property.

The old example for parsing a token looked like this..

```
if token, err := jwt.Parse(tokenString, keyLookupFunc); err == nil {
    fmt.Printf("Token for user %v expires %v", token.Claims["user"], token.Claims["exp"
}
```

is now directly mapped to…

```
if token, err := jwt.Parse(tokenString, keyLookupFunc); err == nil {
    claims := token.Claims.(jwt.MapClaims)
    fmt.Printf("Token for user %v expires %v", claims["user"], claims["exp"])
}
```

`StandardClaims` is designed to be embedded in your custom type. You can supply a custom claims type with the new `ParseWithClaims` function. Here's an example of using a custom claims type.

```
type MyCustomClaims struct {
    User string
    *StandardClaims
}

if token, err := jwt.ParseWithClaims(tokenString, &MyCustomClaims{}, keyLookupFunc); err
    claims := token.Claims.(*MyCustomClaims)
    fmt.Printf("Token for user %v expires %v", claims.User, claims.StandardClaims.Expire
}
```

`ParseFromRequest` has been moved

To keep this library focused on the tokens without becoming overburdened with complex request processing logic, `ParseFromRequest` and its new companion

`ParseFromRequestWithClaims` have been moved to a subpackage, `request`. The method signatues have also been augmented to receive a new argument: `Extractor`.

`Extractors` do the work of picking the token string out of a request. The interface is simple and composable.

This simple parsing example:

```
if token, err := jwt.ParseFromRequest(tokenString, req, keyLookupFunc); err == nil {
    fmt.Printf("Token for user %v expires %v", token.Claims["user"], token.Claims["exp"
}
```

is directly mapped to:

```
if token, err := request.ParseFromRequest(req, request.OAuth2Extractor, keyLookupFunc);
    claims := token.Claims.(jwt.MapClaims)
    fmt.Printf("Token for user %v expires %v", claims["user"], claims["exp"])
}
```

There are several concrete `Extractor` types provided for your convenience:

- `HeaderExtractor` will search a list of headers until one contains content.
- `ArgumentExtractor` will search a list of keys in request query and form arguments until one contains content.
- `MultiExtractor` will try a list of `Extractors` in order until one returns content.
- `AuthorizationHeaderExtractor` will look in the `Authorization` header for a `Bearer` token.
- `OAuth2Extractor` searches the places an OAuth2 token would be specified (per the spec): `Authorization` header and `access_token` argument
- `PostExtractionFilter` wraps an `Extractor`, allowing you to process the content before it's parsed. A simple example is stripping the `Bearer` text from a header

RSA signing methods no longer accept `[]byte` keys

Due to a critical vulnerability, we've decided the convenience of accepting `[]byte` instead of `rsa.PublicKey` or `rsa.PrivateKey` isn't worth the risk of misuse.

To replace this behavior, we've added two helper methods: `ParseRSAPrivateKeyFromPEM(key []byte) (*rsa.PrivateKey, error)` and `ParseRSAPublicKeyFromPEM(key []byte) (*rsa.PublicKey, error)`. These are just simple helpers for unpacking PEM encoded PKCS1 and PKCS8 keys. If your keys are encoded any other way, all you need to do is convert them to the `crypto/rsa` package's types.

```
func keyLookupFunc(*Token) (interface{}, error) {
    // Don't forget to validate the alg is what you expect:
    if _, ok := token.Method.(*jwt.SigningMethodRSA); !ok {
```

```go
        return nil, fmt.Errorf("Unexpected signing method: %v", token.Header["alg"])
    }

    // Look up key
    key, err := lookupPublicKey(token.Header["kid"])
    if err != nil {
        return nil, err
    }

    // Unpack key from PEM encoded PKCS8
    return jwt.ParseRSAPublicKeyFromPEM(key)
}
```