Alexa Skills, Angular Security, Docker, SQL Server Reporting Services

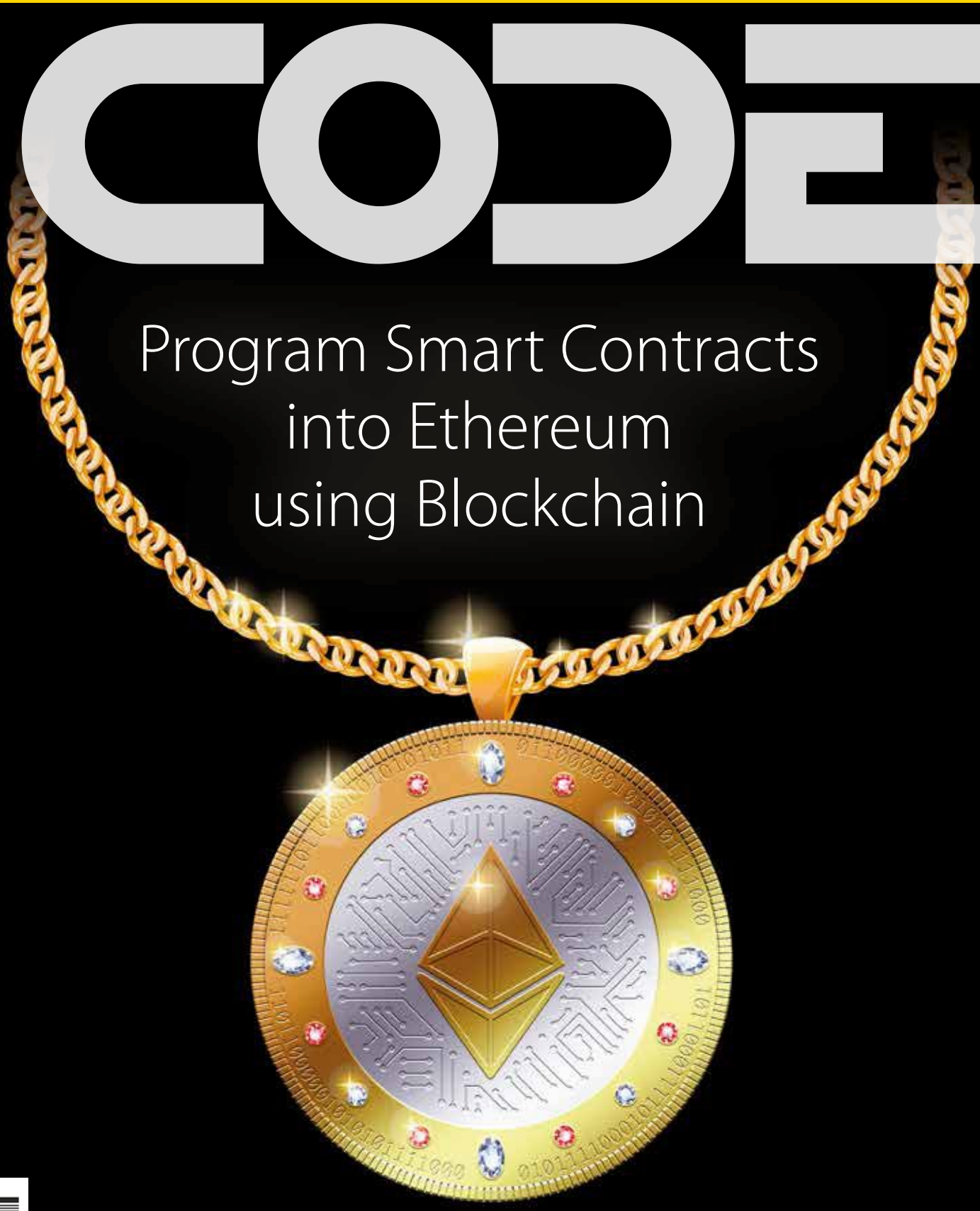
CODE

MAY
JUN
2018

codemag.com - THE LEADING INDEPENDENT DEVELOPER MAGAZINE - US $ 5.95 Can $ 8.95

CODE

Program Smart Contracts into Ethereum using Blockchain



Building Custom Alexa Skills

Building Containers with Docker

Introducing Progressive Web Applications

06
0 74470 56130 9

Features

Columns

Departments

US subscriptions are US $29.99 for one year. Subscriptions outside the US pay US $44.99. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa, and Discover credit cards are accepted. Bill me option is available only for US subscriptions. Back issues are available. For subscription information, send e-mail to subscriptions@codemag.com.

Subscribe online at codemag.com

CODE Component Developer Magazine (ISSN # 1547-5166) is published bimonthly by EPS Software Corporation, 6605 Cypresswood Drive, Suite 300, Spring, TX 77379 U.S.A. POSTMASTER: Send address changes to CODE Component Developer Magazine, 6605 Cypresswood Drive, Suite 300, Spring, TX 77379 U.S.A.

Canadian Subscriptions: Canada Post Agreement Number 7178957. Send change address information and blocks of undeliverable copies to IBC, 7485 Bath Road, Mississauga, ON L4T 4C1, Canada.

LEADTOOLS

One SDK, Multiple Platforms

LEADTOOLS toolkits will help you create desktop, web, server, and mobile applications with the greatest collection of programmer-friendly and cross-platform document, medical, and multimedia technologies.

OCR | FORMS | BARCODE | DICOM | PACS

+ MANY MORE

Get Started Today

DOWNLOAD OUR FREE EVALUATION

LEADTOOLS.COM

A Software "Pre-Mortem"

As you may already realize, many of my editorials spring from various issues that my team and I encounter when building software for our clients. Most of my editorials are post-mortems on our projects or thoughts spawned by the project experience. This time, I'm going to try something a bit different.

I'm going to do a "pre-mortem." What I mean by that is that I'm going to write about a project from its inception. It's likely that I'll continue writing about this project as it progresses. Real world work tends to provide fertile material for my editorials and I expect this project to be no different.

The Project

Dear reader, you're about to embark on another journey into Application Conversion Land. Yes, we're doing another conversion project. This time we're taking a 20-year-old Visual FoxPro project and converting it into a modern architecture using WPF, C#, and SQL Server.

If you follow my editorials, you'll recall that we've been involved in several conversion projects. My Nov/Dev 2017 editorial "Lessons Learned from a Second Trip Around the Block" discussed a number of lessons we learned on a long-term conversion project. These lessons are at the forefront of every member's mind as we embark on this new adventure.

The Team

The most important aspect of a successful software project is the team of people that build the software. By Dash Point Software (my company) standards, we have a reasonably large development team for this project. The team consists of seven software engineers assisted by other team members who handle roles such as project management, testing, and database management.

A critical aspect of building this team is the relationships among the different members. I chose "known quantities." All of the developers had worked on one or more projects together, which helps cut down on the sometimes-slow process of learning how to work with new people and how their skills fit into the team. There's another unique aspect to this familiarity: Some of us were on board 20 years ago when the initial project was created. That longevity is important because it provides continuity and a deep understanding of our problem domain.

The Process

This project is being developed by combining several agile software development techniques, principles, and processes. Agile software practitioners will tell you that there's no one way to manage an agile software development, and I believe that to be true. For this project, I chose to employ the following concepts: pair programming, unit and integration tests, daily standups, and Kanban management.

Pair Programming

One of the most valuable agile techniques is the concept of pair programming. I find this technique valuable because it helps spread knowledge from one team member to another. I've been using Visual Studio and C# from Day One and even so, I still learn new tools and techniques from other developers. We don't practice full time pair programming because our teams are geographically distributed, but we try to emphasize the pairing off for a few hours of every week.

Unit and Integration Tests

Unit tests and integration tests serve a very important role in our application process. These tests are long-term investments in the stability and maintainability of our software. We're building this app to last another 20 years and the automated tests serve to insure the long-term viability of our project.

Daily Standups

When building software with a team of this size, it's important to touch base on a frequent basis. For this, we have daily standups. The goal is to keep other developers informed about what's being worked on by other developers and, most importantly, for the team to communicate any blocking issues they may have.

Kanban Management

We're using a Kanban board to manage all tasks that are under way at any given time. We're starting to build a backlog, insuring that developers have a constant set of items to work on. We're not following Kanban 100%; we're using only the tools that work for us.

The Tools

"I can fix it. My dad is a TV repairman. He has an awesome set of tools." No truer words have been spoken than those of the stoner-philosopher Jeff Spicoli in the classic film "Fast Times at Ridgemont High." Well, we at Dash Point have an awesome set of tools too. Over the years, we've curated a great set of software development tools and utilities. Here are a two we've deployed:

- **Slack:** We use Slack as mechanism for communicating among all of the software team members. We use Slack to keep a persistent record of questions, concepts, ideas, techniques, and other project information that needs to be shared among members of the team. This tool helps prevent the e-mail barrage that generally happens with software projects.
- **ReSharper and ReSharper Templates:** Our internal framework has a set of standardized ways to perform various tasks. We've created a set of templates to automate the basic bootstrapping of these tasks. This removes some of the mundane tasks (boring) that accompany all software projects

Other tools we use are Jira (Kanban Management), Google Sheets (task management overview), Jenkins (continuous integration server), and SQL Prompt (a SQL standardization and formatting tool).

Early Lessons

A few weeks ago, we convened the first meeting; the entire team was present in the same geographic location. The purpose of this meeting was to discuss project goals and software development processes, and to demonstrate how to use the current set of tools and framework code. I can say with zero hesitation that this is one of the best teams I've had the privilege to work with. It took no time for the team to throw out constructive criticism, new ideas, techniques for making our process work better, and especially important, the desire to "do this right" from the start. I really look forward to seeing how we perform over the long haul. From what I'm seeing, we are off to a great start.

Rod Paddock

Learn, Explore, Use

Your Destination for Data Cleansing & Enrichment APIs

Your centralized portal to discover our tools, code snippets and examples.

RAPID APPLICATION DEVELOPMENT

Convenient access to Melissa APIs to solve problems with ease and scalability.

REAL-TIME & BATCH PROCESSING

Ideal for web forms and call center applications, plus batch processing for database cleanup.

TRY OR BUY

Easy payment options to free funds for core business operations.

FLEXIBLE CLOUD APIS

Supports REST, JSON, XML and SOAP for easy integration into your application.

Turn Data into Success – Start Developing Today!

Melissa.com/developer

1-800-MELISSA

melissa

Security in Angular: Part 1

In most business applications, you're going to want to disable, or make invisible, various features such as menu items, buttons, and other UI items, based on who's logged in and what roles or permissions they have. Angular doesn't have anything built-in to help you with this, so you must create it yourself. There are two aspects of security to worry about with Angular applications.

Paul D. Sheriff
http://www.fairwaytech.com

Paul D. Sheriff is a Business Solutions Architect with Fairway Technologies, Inc. Fairway Technologies is a premier provider of expert technology consulting and software development services, helping leading firms convert requirements into top-quality results. Paul is also a Pluralsight author. Check out his videos at http://www.pluralsight.com/author/paul-sheriff.

First, you must develop client-side security objects to control access to your HTML UI elements. Second, you must secure your Web API calls to ensure that only authorized persons can access them. In this series of articles, you're going to be introduced to a few different methods of securing the client-side objects. For the Web API, you're going to use JSON Web Tokens to secure your method calls.

Approaches to Security

There are many different approaches that you can take to securing HTML items in Angular. You can create a simple security object that has one property for each item in your application that you wish to secure, as illustrated in **Figure 1**. This approach is great for small Angular applications, as you won't have many items to secure. For large Angular applications, you'll want to employ a claims-based and/or a role-based solution.

This first article focuses on the simple security object with one property for each item to secure. This approach helps you focus on how to accomplish security before you tackle claims and roles. This article uses mock security objects, so you don't need to use any Web API calls. You're going to learn to retrieve security objects from a Web API in the next article.

Preparing for This Article

To demonstrate how to apply security to an Angular application, I created a sample application with a few pages to display a list of products, display a single product, and display a list of product categories. You can download the sample from the CODE Magazine page associated with this article. You can also download this sample from http://pdsa.com/downloads; select "PDSA/Fairway Articles" from the Category drop-down, then choose "CODE Magazine: Security in Angular - Part 1".

This article assumes that you have the following tools installed:

- Visual Studio Code
- Node
- Node Package Manager (npm)
- Angular CLI

A Look at the Sample Application

There are two menus in the sample you downloaded, Products and Categories (**Figure 2**), that you may wish to turn off, based on permissions assigned to a user. On the product and category list page (**Figure 2**), you may want to turn off the Add New Product button based on permissions, too.

On the product detail page (**Figure 3**), the Save button may be something you wish to turn off. Perhaps you want to let someone view product details but not modify the data.

> Perhaps you want to let someone view product details but not modify the data.

Finally, on the Categories page (**Figure 4**), you may wish to make the Add New Category button invisible for some users.

Create User Security Classes

To secure an application, you need a couple of classes to hold user information. A user class holds the user name and password entered on a log in page. After these values are entered on a log in page, they're passed to a security service class to perform the verification of the user name and password. In this article, a mock array of logins is used to validate against. A user authorization class is returned from the security service class after the verification of the user. This authorization class has one property for each item in your application that you wish to secure. You bind the properties in this class to turn on and off different menus, buttons, or other UI elements on your pages.

User Class

Let's start creating each of these security classes. Open Visual Studio Code and load the sample project that you downloaded. Create the user class to hold the user name and password that the user types into a log in page. Right mouse-click on the \src\app folder and add a new folder named **security**. Right mouse-click on the new security folder and add a file named **app-user.ts**. Add two properties into this AppUser class, as shown in the following code:

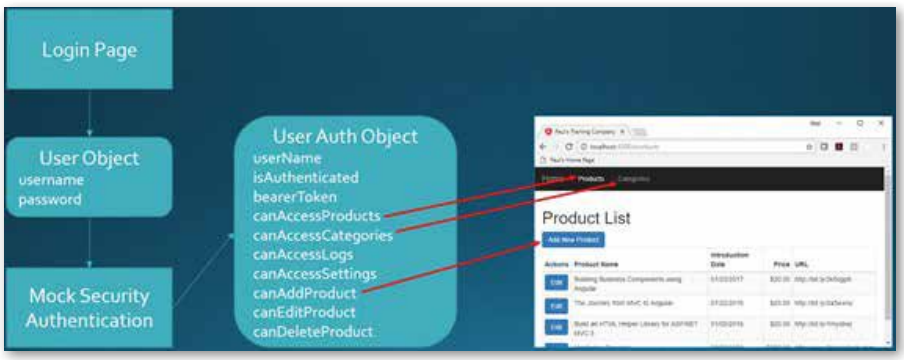


Figure 1: Security authentication and authorization using single properties

```
export class AppUser  {
  userName: string = "";
  password: string = "";
}
```

User Authorization Class

It's now time to create the authorization class to turn menus and buttons off and on. Right mouse-click on the security folder and add a new file named **app-user-auth.ts.** This class contains the **username** property to hold the user name of the authenticated user, a **bearerToken** to be used when interacting with Web API calls, and a Boolean property named **isAuthenticated**, which is only set to true when a user has been authenticated. The rest of the Boolean properties contained in this class are specific for each menu and button you wish to secure.

```
export class AppUserAuth {
  userName: string = "";
  bearerToken: string = "";
  isAuthenticated: boolean = false;
  canAccessProducts: boolean = false;
  canAddProduct: boolean = false;
  canSaveProduct: boolean = false;
  canAccessCategories: boolean = false;
  canAddCategory: boolean = false;
}
```

Login Mocks

In this article, keep all authentication and authorization local within this Angular application. Create a file with an array of mock logins. Right mouse-click on the security folder and add a new file named **login-mocks.ts**. Create a constant named **LOGIN_MOCKS** that's an array of AppUserAuth objects (see **Listing 1**). Create a couple of literal objects to simulate two different user objects that you might retrieve from a database on a back-end server.

Security Service

Angular is all about services, so it makes sense that you create a security service class to authenticate a user and return the user's authorization object with all of the appropriate properties set. Open a VS Code terminal window and type in the following command to generate a service class named **SecurityService**. Add the **-m** option to register this service in the **app.module** file.

```
ng g s security/security --flat -m app.module
```

Open the generated **security.service.ts** file and add the following import statements.

```
import { Observable } from 'rxjs/Observable';
import { of } from 'rxjs/observable/of';

import { AppUserAuth } from './app-user-auth';
import { AppUser } from './app-user';
import { LOGIN_MOCKS } from './login-mocks';
```

Add a property to the SecurityService class to hold the user authorization object. Initialize this object to a new instance of the AppUserAuth class so it creates the object in memory.

```
securityObject: AppUserAuth = new AppUserAuth();
```

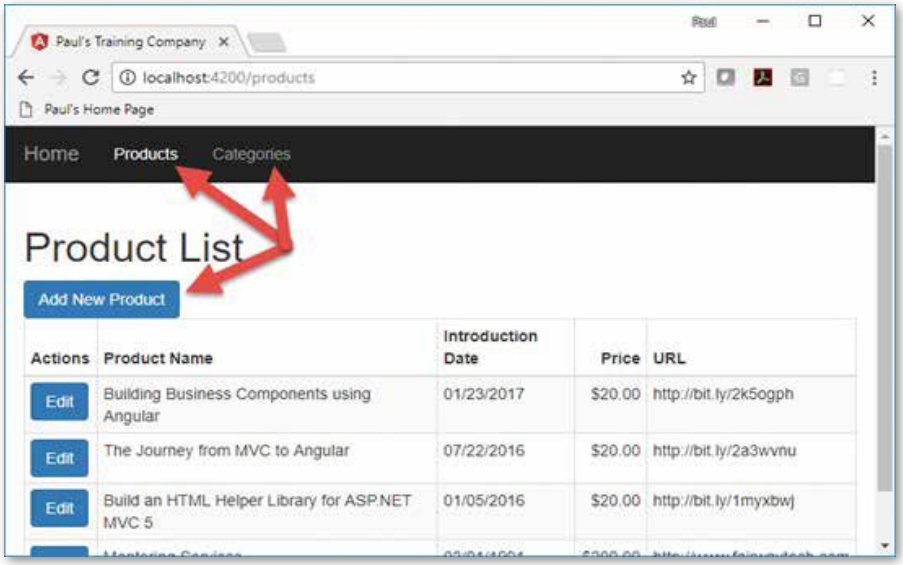


Figure 2: Product list page

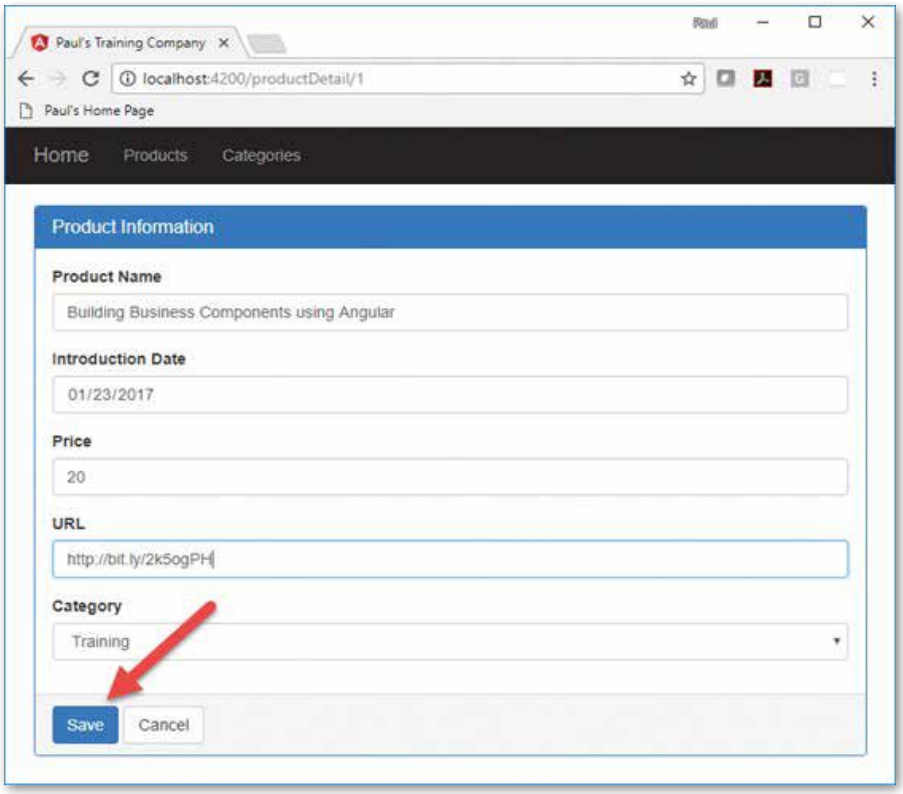


Figure 3: Turn off the Save button based on permissions.

Reset Security Object Method

The **securityObject** property is going to be bound to other security objects in your application. Because of this, you don't want to ever set this property to a new instance of the AppUserAuth, as this can cause those bindings to be released. Instead, just change the properties of this object based on a new user who logs in. Add a method to reset the properties of this security object to default values. Also, notice in this code that you're removing the bearer token property from local storage. You're going to learn why this is necessary in the next article.

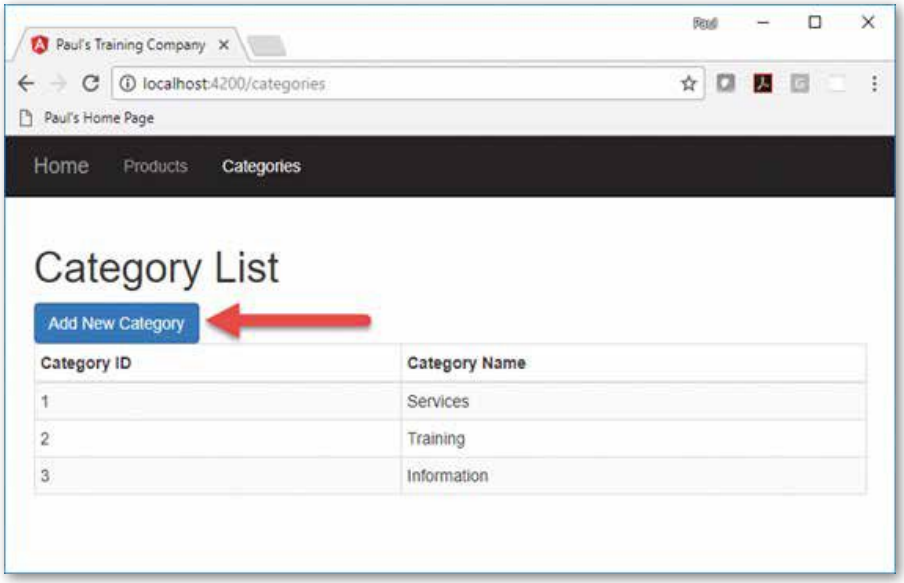


Figure 4: Turn off the Add New Category button based on permissions

```
resetSecurityObject(): void {
  this.securityObject.userName = "";
  this.securityObject.bearerToken = "";
  this.securityObject.isAuthenticated = false;

  this.securityObject.canAccessProducts = false;
  this.securityObject.canAddProduct = false;
  this.securityObject.canSaveProduct = false;
  this.securityObject.canAccessCategories
    = false;
  this.securityObject.canAddCategory = false;

  localStorage.removeItem("bearerToken");
}
```

Login Method

In the next section of this article, you're going to create a log in component. That log in component creates an instance of the AppUser class and binds to the properties of the input fields on the log in page. Once the user has typed in a user name and password, this instance of the AppUser class is going to be passed to a **login()** method in the SecurityService class to determine if the user exists. If the user exists, the appropriate properties are filled into a AppUserAuth object and returned from the login() method.

```
login(entity: AppUser):
     Observable<AppUserAuth> {
  this.resetSecurityObject();

  Object.assign(this.securityObject,
    LOGIN_MOCKS.find(
       user => user.userName.toLowerCase() ===
               entity.userName.toLowerCase()));
  if (this.securityObject.userName !== "") {
    localStorage.setItem("bearerToken",
       this.securityObject.bearerToken);
  }

  return of<AppUserAuth>(this.securityObject);
}
```

In the code above, the first thing that happens is that you reset the security object. Next, use the Object.assign() method to replace all the properties in the **securityObject** property with the properties from the AppUserAuth object returned from the find() method on the **LOGIN_MOCKS** array. If the user is found, the bearer token is stored into local storage. This is for when you need to pass this value to the Web API. My next article will cover the reason for storing this token into local storage.

> My next article will cover the reason for storing this token into local storage.

Logout Method

If you have a login() method, you should always have a logout() method. The logout() method resets the properties in the **securityObject** property to empty fields, or false values. By resetting the properties, any bound properties, such as menus, reread those properties and may change their state from visible to invisible.

```
logout(): void {
  this.resetSecurityObject();
}
```

Login Page

Now that you have a security service to perform a log in, you need to retrieve a user name and password from the

Listing 1: Create a class of mock login objects

```
import { AppUserAuth } from "./app-user-auth";

export const LOGIN_MOCKS: AppUserAuth[] = [
  {
    userName: "PSheriff",
    bearerToken: "abi393kdkd9393ikd",
    isAuthenticated: true,
    canAccessProducts: true,
    canAddProduct: true,
    canSaveProduct: true,
    canAccessCategories: true,
    canAddCategory: false
  },
  {
    userName: "BJones",
    bearerToken: "sd9f923k3kdmcjkhd",
    isAuthenticated: true,
    canAccessProducts: false,
    canAddProduct: false,
    canSaveProduct: false,
    canAccessCategories: true,
    canAddCategory: true
  }
];
```

Listing 1: Create a class of mock login objects Listing 2: Create a login page to bind to the user name and password properties.

```
<div class="row">
  <div class="col-xs-12">
    <div class="alert alert-danger"
    *ngIf="securityObject &&
          !securityObject.isAuthenticated">
      <p>Invalid User Name/Password.</p>
    </div>
  </div>
</div>

<!-- TEMPORARY CODE TO VIEW SECURITY OBJECT -->
<div class="row">
  <div class="col-xs-12">
    <label>{{securityObject | json}}</label>
  </div>
</div>

<form>
  <div class="row">
    <div class="col-xs-12 col-sm-6">
      <div class="panel panel-primary">
        <div class="panel-heading">
          <h3 class="panel-title">Log in</h3>
        </div>
        <div class="panel-body">
          <div class="form-group">
            <label for="userName">User Name
            </label>
            <div class="input-group">
              <input id="userName" name="userName"
                     class="form-control" required
                     [(ngModel)]="user.userName"
                     autofocus="autofocus" />
              <span class="input-group-addon">
                <i class="glyphicon
                          glyphicon-envelope">
                </i>
              </span>
            </div>
          </div>
          <div class="form-group">
            <label for="password">Password</label>
            <div class="input-group">
              <input id="password" name="password"
                     class="form-control" required
                     [(ngModel)]="user.password"
                     type="password" />
              <span class="input-group-addon">
                <i class="glyphicon glyphicon-lock">
                </i>
              </span>
            </div>
          </div>
        </div>
        <div class="panel-footer">
          <button class="btn btn-primary"
                  (click)="login()">
            Login
          </button>
        </div>
      </div>
    </div>
  </div>
</form>
```

user. Create a log in page by opening a terminal window and type in the following command to generate a log in page and component.

```
ng g c security/login --flat -m app.module
```

Open the **login.component.html** file and delete the HTML that was generated. Create three distinct rows on the new log in page (see **Listing 2**):

- A row to display "Invalid User Name/Password"
- A row to display the instance of the securityObject property
- A panel for entering user name and password

Use Bootstrap styles to create each of these rows on this log in page. The first div tag contains a *ngIf directive to only display the message if the **securityObject** exists and the **isAuthenticated** property is false. The second div tag contains a binding to the **securityObject** property. This object is sent to the JSON pipe to display the object as a string within a label. You don't need this row for your final application, but it's useful while debugging to see the values returned in the user authorization class. The last row is a Bootstrap panel into which you place the appropriate user name and password input fields.

Modify Login Component TypeScript

As you can see from the HTML you entered into the login.component.html file, there are two properties required for binding to the HTML elements: user and securityObject. Open the **login.component.ts** file and add the following import statements, or if you wish, use VS Code to insert them for you as you add each class.

```
import { AppUser } from './app-user';
import { AppUserAuth }
  from './app-user-auth';
import { SecurityService }
  from './security.service';
```

Add two properties to hold the user and the user authorization objects.

```
user: AppUser = new AppUser();
securityObject: AppUserAuth = null;
```

To set the securityObject property, you need to inject the SecurityService into this class. Modify the constructor to inject the SecurityService.

```
constructor
  (private securityService: SecurityService) { }
```

The button in the footer area of the Bootstrap panel binds the click event to a method named login(). Add this login() method as shown here.

```
login() {
  this.securityService.login(this.user)
    .subscribe(resp => {
      this.securityObject = resp;
    });
}
```

The login() method on the SecurityService class is subscribed to, and the response that's returned is assigned into the securityObject property defined in this log in component. This is done only for the purpose of debugging. Upon successful return of the login() method, you'd most likely redirect to a home page. Later in this article, you'll learn how to redirect.

Secure Menus

Now that you have the log in working and a valid security object, you need to bind this security object to the main menu. The menu system is created in the **app.component.html** file. Open that file and add the HTML in the next snippet below the closing tag used to create the other menus. This HTML creates a right-justified menu that displays the word "Login" when the user is not yet authenticated. Once authenticated, the menu changes to Logout <User Name>.

```
<ul class="nav navbar-nav navbar-right">
  <li>
    <a routerLink="login"
       *ngIf="!securityObject.isAuthenticated">
      Login
    </a>
    <a href="#" (onclick)="logout()"
       *ngIf="securityObject.isAuthenticated">
      Logout {{securityObject.userName}}
    </a>
  </li>
</ul>
```

Modify the other two menu items on this page to check the security object to determine if they need to be displayed or not. Use the *ngIf directive to check the securityObject property you're going to add to the AppComponent class. Now you see how the Boolean properties you added in the user authorization class correspond to each HTML element you wish to control.

```
<li>
  <a routerLink="/products"
     *ngIf="securityObject.canAccessProducts">
    Products</a>
</li>
<li>
  <a routerLink="/categories"
     *ngIf="securityObject.canAccessCategories">
  Categories</a>
</li>
```

Modify the AppComponent Class

As you saw from the HTML you entered, you need to add the securityObject property to the component associated with this page. Open the **app.component.ts** file and add the securityObject property. Set it equal to a null value to start with so the Invalid User Name/Password message doesn't show.

```
securityObject: AppUserAuth = null;
```

Modify the constructor of the AppComponent class to inject the SecurityService and assign the securityObject property to the property you just created.

```
constructor
  (private securityService: SecurityService) {

  this.securityObject =
     securityService.securityObject;
```


Advertisers Index

```
}
```

Add a logout() method in this class to call the logout() method on the security service class. This method is bound to the click event on the Logout menu item you added in the HTML.

```
logout(): void {
  this.securityService.logout();
}
```

Add Login Route

To get to the login page, you need to add a route. Open the **app-routing.module.ts** file and add a new route like the one shown next.

```
{
  path: 'login',
  component: LoginComponent
},
```

Try It Out

Save all the changes you've made so far. Start the application using **npm start**. Click the Login menu, log in with "psheriff", and notice the properties that are set in the returned security object. Click the log out button, then log back in as "bjones" and notice that different properties are set, and the Product menu disappears. This is because the **canAccessProducts** property in the LOGIN_MOCKS array for BJones is set to false.

Open the **logins-mock.ts** file and set the canAccessProducts property to true for the "BJones" object. You're going to try out some of the different authorization properties later in this article, so this property needs to be set to true to try them out.

```
{
  userName: "BJones",
  bearerToken: "sd9f923k3kdmcjkhd",
  isAuthenticated: true,
  canAccessProducts: true,
  canAddProduct: false,
  canSaveProduct: false,
  canAccessCategories: true,
  canAddCategory: true
}
```

Secure Buttons

In addition to the permissions you added to the menus, you might also want to apply the same to buttons that perform actions. For example, adding a new product or category or saving product data. For this article, you're only learning how to hide HTML elements. If there were Web API method calls behind these buttons, those are not being secured here. You need to secure the Web API using some sort of token system. Those techniques will be covered in the next article.

Secure the Add New Product button by using the security object created after logging in. Open the **product-list.component.html** file and modify the Add New Product button to look like the following:

```
<button class="btn btn-primary"
        (click)="addProduct()"
        *ngIf="securityObject.canAddProduct">
  Add New Product
</button>
```

Open the **product-list.component.ts** file and add a property named securityObject that is of the type AppUserAuth. You're going to want to add this same property to any component in which you wish to use security.

```
securityObject: AppUserAuth = null;
```

Assign the securityObject property you just created to the securityObject property in the SecurityService class. Inject the service in the constructor and retrieve the security object.

```
constructor
 (private productService: ProductService,
  private router: Router,
  private securityService: SecurityService) {

    this.securityObject =
       securityService.securityObject;

}
```

Open the **product-detail.component.html** file and modify the Save button to bind to the canSaveProduct property on the securityObject property. The *ngIf directive causes the button to disappear if the canSaveProduct property is false.

```
<button class="btn btn-primary"
       (click)="saveData()"
       *ngIf="securityObject.canSaveProduct">
  Save
</button>
```

Open the **product-detail.component.ts** file and add the securityObject property, just as you did in the product-list.component.ts file.

```
securityObject: AppUserAuth = null;
```

Modify the constructor to inject the SecurityService and to assign the securityObject property from the SecurityService to the securityObject property you just created in this class.

```
constructor
 (private categoryService: CategoryService,
  private productService: ProductService,
  private route: ActivatedRoute,
  private location: Location,
  private securityService: SecurityService) {

  this.securityObject =
    securityService.securityObject;

}
```

Open the **category-list.component.html** file and modify the Add New Category button to bind to the canAddCategory property on the securityObject property.

Sample Code

You can download the sample code for this article by visiting my website at http://www.pdsa.com/downloads. Select PDSA/Fairway Articles, and then select "CODE Magazine: Security in Angular - Part 1" from the drop-down list. You can also get the sample code from the CODE Magazine page related to this article.

Listing 3: Secure routes using a Route Guard.

```
{
  path: 'products',
  component: ProductListComponent,
  canActivate: [AuthGuard],
  data: {claimName: 'canAccessProducts'}
},
{
  path: 'productDetail/:id',
  component: ProductDetailComponent,
  canActivate: [AuthGuard],
  data: {claimName: 'canAccessProducts'}
},
{
  path: 'categories',
  component: CategoryListComponent,
  canActivate: [AuthGuard],
  data: {claimName: 'canAccessCategories'}
},
```

Only for Small Angular Applications

The downside of using a single property for each element you wish to secure is that you need to add the securityObject property and inject the security service into each component that you wish to secure. This is a lot of code to add to a lot of different components.

That's why this approach is good for small applications but would be somewhat cumbersome to use on larger applications. A better approach is to use a custom structural directive that can look up a property in an array of claims or roles. In a future article, you're going to learn to create a structural directive like this.

```
<button class="btn btn-primary"
        (onclick)="addCategory()"
        *ngIf="securityObject.canAddCategory">
  Add New Category
</button>
```

Open the **category-list.component.ts** file and add the securityObject property.

```
securityObject: AppUserAuth = null;
```

Modify the constructor to inject the SecurityService and to assign the securityObject property from the SecurityService to the securityObject property you just created in this class.

```
constructor
 (private categoryService: CategoryService,
  private securityService: SecurityService) {

  this.securityObject =
    securityService.securityObject;

}
```

Try It Out

Save all the changes you've made and go to your browser. Click the Login menu, log in as the user "psheriff", and notice the properties that are set in the returned security object. Open the Products page and you can click on the Add New Product button. If you click on an Edit button next to one of the products, you can see the Save button on the product detail page. Open the Category page and notice that the Add New Category button isn't visible to you.

> Notice the properties that are set in the returned security object.

Click the logout menu, then log in as the "bjones" user. Notice that different properties are set on the security object. Open the Products page and notice the Add New Product button isn't visible. If you click on an Edit button next to one of the products, the Save button on the product detail page isn't visible to you. Open the Category page and notice that the Add New Category button is visible.

Secure Routes Using a Guard

Even though you can control the visibility of menu items, just because you can't click on them doesn't mean you can't get to the route. You can type the route directly into the browser address bar and you can get to the products page even if you don't have the canAccessProducts property set to true.

To protect the route, you need to build a Route Guard. A Route Guard is a special class in Angular to determine if a page can be activated or deactivated. Let's learn how to build a CanActivate guard. Open a terminal and create a new guard named AuthGuard.

```
ng g g security/auth --flat -m app.module
```

To protect a route, open the **app-routing.module.ts** file and add the canActivate property to those paths you wish to secure, as shown in **Listing 3**. The canActivate property is an array, so you may pass one or multiple guards to this property. In this case, add the AuthGuard class to the array of guards. For each route, you also need to specify the name of the property to check on the security object that's associated with this route. Add a data property, pass in a property named claimName, and set the value of that property to the name of the property associated with the route. This data property is passed to each Guard listed in the canActivate property.

Authorization Guard

Let's write the appropriate code in the AuthGuard to secure the route. As you're going to need to access the property passed in via the data property, open the **auth-guard.ts** file and add a constructor to inject the SecurityService.

```
constructor
  (private securityService: SecurityService) { }
```

Modify the canActivate() method to retrieve the claimName property in the data property. Remove the "return true" statement and add the following lines of code in its place:

```
canActivate(next: ActivatedRouteSnapshot,
            state: RouterStateSnapshot):
    Observable<boolean> | Promise<boolean>
    | boolean {
  // Get property name to check
  let claimName: string =
      next.data["claimName"];

  return this.securityService
      .securityObject.isAuthenticated &&
    this.securityService
      .securityObject[claimName];
}
```

Retrieve the property name to check on the security object using the next parameter. This property is an ActivatedRouteSnapshot and contains the data object passed via the route you created earlier. A true value returned from this guard means that the user has the right to navigate to this route. Check to ensure that the isAuthenticated property on the securityObject is a true value and that the property name passed in the data object is also a true value.

Try It Out

Save all the changes you have made, go to the browser, and type directly into the browser address bar: http://localhost:4200/products. If you're not logged in, you're not able to get to the products page. Your guard is working; however, it ends up displaying a blank page. It would be better to redirect to the log in page.

Redirect to the Log in Page

To redirect to the log in page, modify the AuthGuard class to perform the redirection if the user isn't authorized for the current route. Open the **auth-guard.ts** file and inject the Router service into the constructor.

```
constructor
  (private securityService: SecurityService,
   private router: Router) { }
```

Locate the canActivate() method, remove the current return statement, and replace it with the following lines of code.

```
if (this.securityService
      .securityObject.isAuthenticated
    && this.securityService
      .securityObject[claimName]) {
  return true;
}
else {
  this.router.navigate(['login'],
     { queryParams: { returnUrl: state.url } });
  return false;
}
```

If the user is authenticated and authorized, the Guard returns a true and Angular goes to the route. Otherwise, use the Router object to navigate to the log in page. Pass the current route the user was attempting to get to as a query parameter. This places the route on the address bar that the login component retrieves and uses to go to the route requested after a valid log in.

Try It Out

Save all your changes, go to the browser, and type directly into the browser address bar: http://localhost:4200/products. The page resets, and you're directed to the login page. You should see a returnUrl parameter in the address bar. You can log in, but you won't be redirected to the products page; you need to add some code to the log in component first.

Redirect Back to Requested Page

If the user logs in with the appropriate credentials that allows them to get to the requested page, you want to direct them to that page after log in. The LoginComponent class should return the returnUrl query parameter and attempt to navigate to that route after successful log in. Open the **login.component.ts** file and inject the ActivatedRoute and the Router objects into the constructor.

```
constructor
  (private securityService: SecurityService,
   private route: ActivatedRoute,
   private router: Router) { }
```

Add a property to this class to hold the return URL if any is retrieved from the address bar.

```
returnUrl: string;
```

Add a line to the ngOnInit() method to retrieve this *returnUrl* query parameter. If you click on the Login menu directly, the queryParamMap.get() method returns a null.

```
ngOnInit() {
  this.returnUrl =
     this.route.snapshot
       .queryParamMap.get('returnUrl');
}
```

Locate the login() method and add code after setting the securityObject to test for a valid URL and to redirect to that route if there is one.

```
login() {
  this.securityService.login(this.user)
    .subscribe(resp => {
      this.securityObject = resp;
      if (this.returnUrl) {
        this.router
          .navigateByUrl(this.returnUrl);
      }
    });
}
```

Try It Out

Save all your changes, go to the browser, and type directly into the browser address bar: http://localhost:4200/products. You'll be directed to the log in page. Log in as "psheriff" and you're redirected to the products list page.

Summary

In this article, you learned to add client-side security to your Angular applications. Using a class with properties to represent each permission you want to grant to each user makes securing menu links and buttons easy. Apply Route Guards to your routes to ensure that no one can get to a page by typing directly into the address bar. Everything was done client-side in this article, but you can authenticate users and return a security authorization object using a Web API call. You'll learn how to connect to a Web API in the next article in this series on Angular security.

Paul D. Sheriff

Identify faces with Microsoft Cognitive Services

Microsoft Cognitive Services, or as I like to call it, AI for the common folk, allows your applications to hear, see, and understand just like you would. Think of all the people you know, your best friends, your family, your boss, people you dislike; are you thinking of their faces? When your boss knocks on your cubicle with a coffee cup in his hand, and says, "Hey Peter," you instantly know if you'll need to work the weekend. You see his emotions in his face.

Sahil Malik
www.winsmarts.com
@sahilmalik

Sahil Malik is a Microsoft MVP, INETA speaker, a .NET author, consultant and trainer.

Sahil loves interacting with fellow geeks in real time. His talks and trainings are full of humor and practical nuggets. You can find more about his training at http://www.winsmarts.com/training.aspx.

His areas of expertise are cross platform Mobile app development, Microsoft anything, and security and identity.

Microsoft Cognitive Services puts all that power into your applications. And it's easy! If you can call a REST API, your application can do amazing things. Although the focus of this article is on a very narrow capability, recognizing faces, let me tell you some other amazing capabilities Cognitive Services has.

Microsoft Cognitive Services: The Vision API

Microsoft Cognitive Services is comprised of Vision, Speech, Knowledge, Search, and Language APIs. Within each of these buckets, there are numerous capabilities.

The Vision API allows you to distill actionable information from images. This means that you show it a picture, and it tells you what it sees in the picture: it describes it back to you. Yes really. You show it a picture of a dog, and it'll tell you "Hey, this is a dog." But it gets better. It can also tell you if the picture's black and white, or what the various important colors are in it. It can do handwriting or OCR recognition. It can differentiate among line art, clip art, or photographs. It can flag adult or racy content, generate thumbnails, detect faces, even identify emotions on the faces. It can even see an image and recognize celebrities or landmarks, or you can create your own models, say dog breeds, and it recognizes what breed a dog is, by showing it a picture of a dog! Any dog!

In this article, I'll show the Vision API a picture of a person, and it will recognize who the person is. Specifically, the Face API part of the Vision API will do this.

The speech and language APIs are somewhat interconnected. They can do things like speech to text and text to speech. They can even do things like recognize a language if you provide some input text. Or detect sentiment, key phrases, and break apart sentences if you provide some input text. It can even do translations between languages. Pair these together and you can do real-time text-to-speech in one language, translate to another, and text-to-speech in another language. These are things you can build, easily, if you can call a REST API. Oh, did I mention that it can also do spell check? I know you're laughing—haven't we had spell check for ages in Microsoft Word? Yes, but powered by the cloud, it's far more sophisticated.

Perhaps the most interesting part of the speech and language APIs is LUIS, or Language Understanding Intelligent Service. It allows your applications to understand natural language as input and derive actionable information out of them. For instance, wash my car, or clean my car, are about the same thing, aren't they? You can pair this with, say, the bot framework, and speech-to-text, and you now have a completely new level of application, specific to your domain. Imagine building the next Alexa/Cortana/Siri for your business applications. Would you like to walk into a meeting room and ask, "Is this meeting room free?"

Building these kinds of systems is easy! And you can write it in any platform you wish. You could build it in Node.js and run it on a $39 Arduino.

There's so much that you can build easily with Cognitive Services. I hope to talk about many such capabilities in upcoming articles, but for now, let me return to the focus of this article, identifying faces.

Identifying vs. Recognizing

Let's start by defining the problem domain. Cognitive Services allow you to easily recognize faces in a picture. You show it a picture, and it tells you that at certain rectangle position, there is a face. It'll even tell you things like this person is a female, age 27, who is 99% happy, 1% angry, 0.9% surprised, with a head tilted in a certain way, is bald or not, wears eyeglasses, and a lot more!

As impressive as that is, that's easy! In this article, I'll go the next step: I'll first build a library of people with associated faces. For instance, I'll show former president Obama, with pictures of him. And there's another person, and yet another person, and hey look, yet another person. You could have literally thousands of faces in a library, but I'll keep it demo level at three people. Then I'll input a picture that the program has never seen before. This picture is of a person, a person who is in our library. Or it could be a picture with more than one person, or a picture where one of the people in the picture is part of our library. And the computer will "identify" the person, as in "I think this is Barack Obama, and my confidence is 99%."

Before we get rolling, let's understand some basic theory first.

How Identification Works

In order to perform identification, you first need to create a **Person** group. This is your dataset of people among which recognition will be performed.

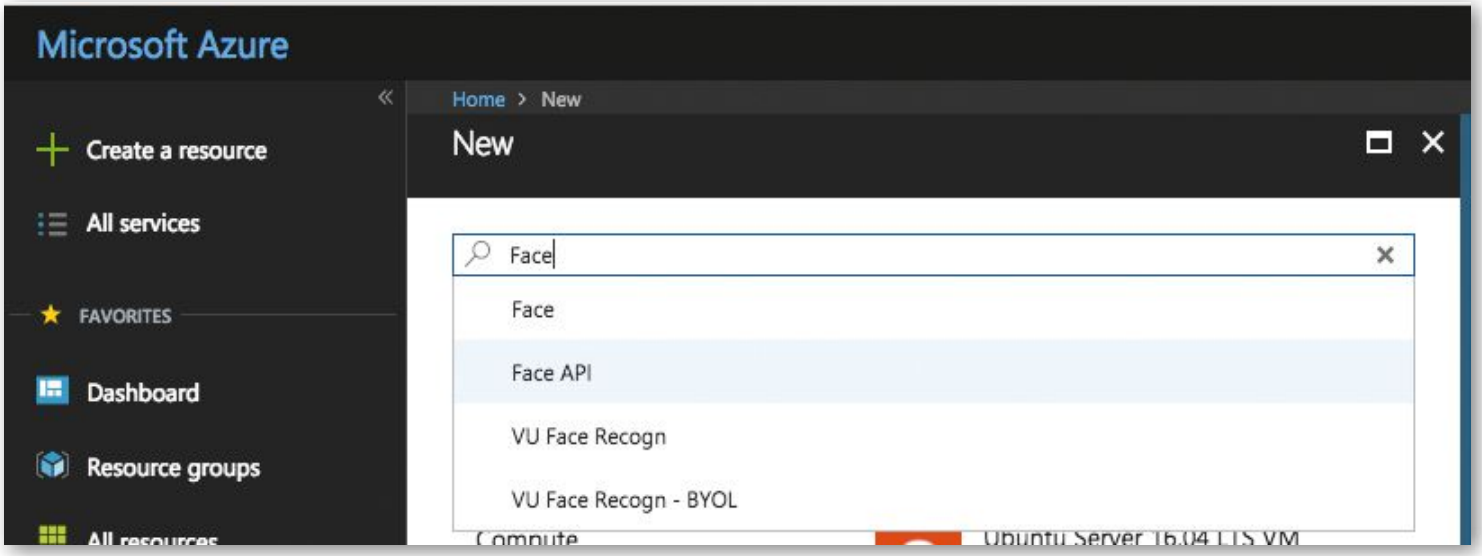


Figure 1: Create an instance of the Face API.

Within that **Person Group**, you add numerous images of people. The number of Persons has a subscription-level limit and a Person Group level limit. Free tier subscriptions have a limit of 1,000 persons per Person Group and 1,000 persons total per subscription. The S0-level tier subscriptions have these limits: 10,000 Persons per Person Group, 100M Persons total and 1M Person Groups per subscription. My demo will have three persons, so I think I can get by with the free tier. But think about it; 100 million persons. Imagine the power this puts in your hands.

Once you have created the Persons Group and the list of Persons, the next thing you do is add faces to each person. This is simply a matter of uploading pictures of each Person with a clear view of their faces. The pictures need to meet the following criteria:

- Each person has a maximum of 248 faces—that's 248 versions of the same person.
- The image format must be JPEG, PNG, GIF (the first frame), or BMP.
- The image file size should be larger than or equal to 1KB but no larger than 4MB.
- The detectable face size is between 36x36 to 4096x4096 pixels. The faces out of this range will not be detected.
- You can submit a picture with many faces, but you must specify the rectangle where the specific person of interest is. The rectangle specified by targetFace should contain exactly one face. Zero or multiple faces will be regarded as an error.
- Out of detectable range face size, large head-pose, or very large occlusions also result in fail to add a person's face.

Easy enough! I think I can find plenty of pictures of the former president.

Once you are finished managing persons and person faces, you train the model. Although this sounds about as bad as going to the gym, in this case, it's a REST call.

Listing 1: My config.ts file

```
export let config = {
    face: {
        endPoint:
      "https://westus.api.cognitive.microsoft.com/face/v1.0",
        key1: "<removed>",
        key2: "<removed>"
    }
};
```

Listing 2: Providing the authentication header.

```
function getRequestOptions(): request.CoreOptions {
    return {
        headers: {
            "Content-Type": "application/octet-stream",
            "Ocp-Apim-Subscription-Key": config.face.key1
        }
    };
}
```

Finally, you submit an input image and the AI Gods tell us who this person is! Doesn't that sound easy? Let's get started!

I must mention that given the number of steps, it's not possible for me to show every line of code in this article. However, you can download and set up the finished code. The instructions can be found in the "set up the code" section of this article.

Set Up the Face API in Azure

Like many Cognitive Services' APIs, in order to use the Face API, you need to provision it for use in the Azure Portal. Go ahead and log in to the Azure portal, click on the "Create a resource" link, and choose to create a new instance of the Face API, as shown in **Figure 1**.

Among other things, you'll need to specify a pricing tier. For the purposes of this article, the F0 subscription-level free tier is fine.

Once you've created this resource, grab the endpoint address and the two keys. I chose to put mine into a file called config.ts, as can be seen in **Listing 1**.

Authenticate to the Face API

Calls to the Face API are simple REST calls. They can be GET/POST/DELETE or any other such common HTTP verb. But the calls must be authenticated. To authenticate the call, you need to pass in a specific HTTP header called **Ocp-Apim-Subscription-Key**, and then pass in either of the two keys from **Listing 1**. You might wonder, why two keys? Well, the two keys are equivalent. You can pass either. The reason they provide two is for redundancy. If one key gets compromised, you can choose to use the other while the first one is regenerated.

I'm using a node package called **request** to make REST API calls. The code to provide the authentication header along with a default content type can be seen in **Listing 2**.

Face API Basics

Before I dive into the depths of identifying faces, let's first understand the capabilities of the Face API. As the name suggests, the Face API lets you work with faces, human faces. The focus of this article is recognizing people, but the Face API can do so much more.

Finding Similar Faces

As the name suggests, the Face API can accept an image with a face, or an image with multiple faces, and a single face specified as coordinates, and then given a set of faces, it returns a set of faces that looks the most similar to the target face.

Grouping Faces

Given a set of input faces, the Face API groups them into several groups based on similarity.

Detection

Detection, as the name suggests, detects human faces in an image. But it can do so much more. When you call the **Detect** method, you have a choice of requesting multiple details of the recognized faces. The more details you request, the longer the request takes. In your inputs to the detect method, you can pass in various optional request parameters.

- **returnFaceID**: This is a unique face ID of the detected face created by the detection API. This expires 24 hours after a detection call; you'll see later in this article how this is useful against recognizing faces in multiple sets of Person Groups.
- **returnFaceLandmarks**: Face landmarks are a series of detailed points on a face, typically points of face components like the pupils, canthus, or nose. By default, there are 27 predefined landmark points that provide all of the details of a recognized face, including eyebrows, and all relevant positions to the nose, lips, and eyes.
- **returnFaceAttributes**: This is a comma-separated list of various attributes of the face you'd like to have returned. Supported face attributes include age, gender, headPose, smile, facialHair, glasses, emotion, hair, makeup, occlusion, accessories, blur, exposure, and noise. Yes, you can pass in an image, and the Face API will tell you if the person looks angry or not using the "emotion" attribute. You could write an interesting analysis program to prove that perhaps bald people are usually less angry than hairier people by using the "hair,emotion" face attributes.

Let's see a simple example of attribute detection in a passed-in image. As you can imagine, the call to the Face API is a simple REST call, as shown here:

```
const uri = config.face.endPoint + "/detect?" +
  querystring.stringify(params);
request.post(
    uri,
    requestOptions,
    (err, response, body) => {
        resolve(body);
    }
);
```

The URI is quite straightforward. The magic is in the **requestOptions** variable.

I start by creating the basic request options with the authentication header and using the method shown in **Listing 2,** as follows:

```
const requestOptions = getRequestOptions();
```

Next, I craft up the params querystring. It's in this params variable that I specify what I'd like to have detected. I'll just specify everything; I want landmarks and I want all the attributes.

My params look like this:

```
const params = {
    "returnFaceId": "true",
    "returnFaceLandmarks": "true",
    "returnFaceAttributes": attributes
};
```

And the attributes variable includes all supported attributes, as shown here:

```
let attributes =
    "age,gender,headPose,smile," +
    "facialHair,glasses,emotion," +
    "hair,makeup,occlusion,accessories," +
    "blur,exposure,noise";
```

Next, I specify the contents of an image as the payload of my request. This is easily accomplished using the code shown here:

```
requestOptions.body =
 fileHelpers.readImage(__dirname + "/"
  + fileName);
```

The code for the readImage method can be seen in **Listing 5**.

Now, it's just a matter of calling this method with an example image. This can be seen here:

VENDORS: ADD A REVENUE STREAM BY OFFERING ESCROW TO YOUR CUSTOMERS!

CAN YOU TAKE THE RISK ?

Less than $1 per day!

Affordable High-Tech Digital Escrow

Tower 48 is the most advanced and affordable digital escrow solution available. Designed and built specifically for software and other digital assets, Tower 48 makes escrow inexpensive and hassle free. Better yet, as a vendor, you can turn escrow into a service you offer to your customers and create a new revenue stream for yourself.

Regardless of whether you are a vendor who wants to offer this service to their customers, or whether you are a customer looking for extra protection, visit our web site to start a free and hassle-free trial account or to learn more about our services and digital escrow in general!

Visit **www.Tower48.com** for more information!

TOWER 48

```
faceHelpers.detectFaceWithAttributes(
  './input.jpg')
    .then(results => {
        console.log(results);
    });
```

The input image I am using can be seen a little further on, in **Figure 5**.

Running this code sends back a huge JSON object; after all, we did ask it to analyze quite a bit. At a high level, the JSON object is an array of objects. Why an array? Because there could be multiple faces in the same image. You can analyze up to 64 faces in one call and each face needs to be between 36x36 to 4096x4096 wide.

In each of the objects in the array, you'll find four properties:

- **faceID**: a unique GUID that helps the Face API remember the given face for up to 24 hours
- **faceRectangle:** where in the picture the face appears
- **faceLandmarks:** the various landmarks detected on the face
- **faceAttributes:** an object representing the various attributes analyzed

Of the four, faceID and rectangle are quite straightforward, so let me dive a bit into the landmarks and attributes.

Landmarks

Under landmarks, you'll see numerous properties, each with an x,y coordinate. The following properties are returned:

- pupilLeft, pupilRight
- noseTip, noseRootLeft, noseRootRight, noseLeftAlarTop, noseRightAlarTop, noseLeftAlarOutTip
- mouthLeft, mouthRight
- eyebrowLeftOuter, eyebrowLeftInner, and similar properties for the right eyebrow.
- eyeLeftOuter, eyeLeftTop, eyeLeftBottom, eyeLeftTop, and similar properties for the right eye.
- upperLipTop, upperLipBottom, underLipTop, underLipBottom

As you can see, pretty much the whole face is broken down into landmarks; you can easily zero in on the entire face structure given these values. But wait, there's more, in attributes.

Attributes

Under attributes, you request all the attributes that the Face API detect method supports, and as you can imagine, you get a lot of information back. I encourage you to try this out for yourself, but meanwhile, let me show you some interesting returned properties.

Attributes lets you know if the person is smiling, and by how much:

```
"smile": 0.27,
```

It lets you know exactly how their head is positioned:

```
"headPose": {
    "pitch": 0.0,
    "roll": -5.8,
    "yaw": -18.1
},
```

It tells you the person's gender:

```
"gender": "male",
```

And their age:

```
"age": 53.9,
```

Whether or not they have facial hair and what kind is it:

```
"facialHair": {
    "moustache": 0.0,
    "beard": 0.1,
    "sideburns": 0.0
},
```

And if they are wearing glasses or not:

```
"glasses": "NoGlasses",
```

It even detects the person's emotion and gives it a 0-1 rating on eight different aspects of emotion:

```
"emotion": {
    "anger": 0.0,
    "contempt": 0.006,
    "disgust": 0.001,
    "fear": 0.0,
    "happiness": 0.27,
    "neutral": 0.724,
    "sadness": 0.0,
    "surprise": 0.0
},
```

I can see many uses for this. Imagine that when you unlock your significant other from the car trunk, it would be quite handy to have the emotion API help you out when you need it the most. (Just kidding!)

It also tells you the quality of the picture as blur, exposure, and noise:

```
"blur": {
    "blurLevel": "low",
    "value": 0.06
},
"exposure": {
    "exposureLevel": "goodExposure",
    "value": 0.66
},
"noise": {
    "noiseLevel": "low",
    "value": 0.0
},
```

It tells you if the person is wearing any make up:

```
"makeup": {
    "eyeMakeup": false,
```

Listing 3: Create a Person Group

```
export function createPersonGroup(
   personGroupId: string): Promise<string> {
    const promise = new Promise<string>((resolve, reject) => {
        const requestOptions = getRequestOptions();
        requestOptions.headers['Content-Type'] = 'application/json';
        requestOptions.body = JSON.stringify({
            'name': personGroupId
        });

        request.put(
            config.face.endPoint +
                '/persongroups/' + personGroupId,
            requestOptions,
            (err, response, body) => {
                if (err) { reject(false); }
                else { resolve(personGroupId); }
            }
        )
    });
    return promise;
}
```

```
    "lipMakeup": false
},
```

I, for one am thrilled that our ex-president looks good without make up on, or for that matter without any accessories, which means that there's just an empty array. Accessories include things like earrings, tattoos, and other ornaments.

```
"accessories": [],
```

If you're having a hard time recognizing the person via code, you can programmatically find out if the face is occluded:

```
"occlusion": {
    "foreheadOccluded": false,
    "eyeOccluded": false,
    "mouthOccluded": false
},
```

And finally, Cognitive Services provides you with an analysis of their hair. It returns a JSON object telling you on a scale of 0-1 how bald the person is. The former president is 0.05 bald, if you're interested. And it does an analysis on their hair color as an array of color and confidence, so Obama's hair is gray, with a confidence of 1.0, black with confidence of 0.28, etc.

A Practical Example

The real power of cognitive API is unlocked when you start combining many APIs. Imagine a natural language interactive robot. The kind you speak to and it responds back in a voice. It could look at your face and recognize who you are when you walk into a store. It could recognize your emotion in the way you look or speak. It can understand thousands of languages. And it can understand natural language with the help of LUIS. When it analyzes your face, it can know exactly what emotion you're exhibiting. If you look angry, it knows, and reacts accordingly. If you're happy, it perhaps tries to sell you more stuff.

With Cognitive Services, today, you can build a robot that will:

- Recognize you soon as you walk into the store
- Speak your language
- Talk to you in a natural language
- Tailor its help to exactly how you feel at the moment

Given how easy all of this is, I'm surprised we don't already have it. I'm sure that we will have all of it before the first commercial flying car.

For now, let's get back to the first part of this problem, which is recognizing you soon as you walk into the store.

Creating a Person Group

Creating a Person Group is a matter of issuing a PUT request to a URL that looks like this:

```
config.face.endPoint +
  '/persongroups/' + personGroupId
```

The personGroupId is simply a user-provided string. The valid characters include numbers, English letters in lower case, a hyphen ("-") and an underscore ("_"). The maximum length of the personGroupId is 64. Along with such a request, you need to include a request body with the following JSON object:

```
{'name': personGroupId}
```

The full code for creating the Person Group can be seen in **Listing 3**.

Create a Person

Once you've created a Person Group, the next step is to add Persons into that Person Group. This is a simple post request to the following endpoint:

```
config.face.endPoint +
  '/persongroups/' + personGroupId + '/persons'
```

Note that the request URL includes the personGroupId. This is how you tell the Face API which Person Group a Person belongs in. Also, you need to specify the name of the Person you're adding as a JSON object that looks like this:

```
{'name': personName}
```

The name is the display name of the target person. This can be up to 128 characters. You can also optionally send some user-provided data attached to the Person as a **userData** property on the input JSON object. You can find the full listing for the createPerson method in **Listing 4**.

Listing 4: Create a Person

```
export function createPerson(
  personGroupId: string, personName: string): Promise<string> {
    const promise = new Promise<string>((resolve, reject) => {
        const requestOptions = getRequestOptions();
        requestOptions.headers['Content-Type'] = 'application/json';
        requestOptions.body = JSON.stringify({
            'name': personName
        });

        request.post(
            config.face.endPoint + '/persongroups/'
                + personGroupId + '/persons',
            requestOptions,
            (err, response, body) => {
                if (err) { reject(false); }
                else { resolve(body); }
            }
        )
    });
    return promise;
}
```

Listing 5: Read contents of the image

```
export function readImage(filePath: string) {
    const fileData = fs.readFileSync(filePath).toString("hex");
    const result = [];
    for (let i = 0; i < fileData.length; i += 2) {
        result.push(
          parseInt(fileData[i] + "" + fileData[i + 1], 16))
    }
    return new Buffer(result);
}
```

Add a Person Face

Adding a Person Face is, you guessed it, another REST call. This time it's a POST request to the following URL:

```
config.face.endPoint + '/persongroups/' +
personGroupId + '/persons/' +
personId + '/persistedFaces';
```

As you can see from the URL, you're posting to the /persistedFaces URL for the given person in the given person group. The next question is: How do you specify the actual file contents? There are two ways.

Either you can specify the content-type header to be application/json, and send the following JSON object in the body of the POST request:

```
{url:'url_to_the_image'}
```

Or you can specify the content-type header to be application/octet-stream and send the contents of the image. The method to read the contents of the image can be seen in **Listing 5**.

The code to issue a POST request is quite similar to **Listing 3** and **Listing 4**, so I'll omit that here. You can find the full code for this article in the "Set up the code" section of this article**.**

Train the Model

Once you've created the Person Group, added Persons, and added Faces to the Persons, you need to train the model before you can start asking it to identify people. Training the Person Group is yet another REST call. You simply issue a POST request to the following URL:

```
config.face.endPoint + '/persongroups/'
+ personGroupId + '/train'
```

The important difference here is that the train operation is a long-running operation. You don't immediately get a success or failure response. Instead, you need to call a specific URL to check the status of the training. The URL is here:

```
config.face.endPoint + '/persongroups/'
+ personGroupId + '/training'
```

This request may take a long time to complete, so usually you call this endpoint every few seconds and check the return JSON object. The return JSON object has the following properties:

- **Status:** With values **notstarted**, **running**, **succeeded**, or **failed**
- **createdDateTime:** A UTC time informing you of when the Person Group was created
- **lastActionDateTIme:** A UTC time informing you of the last time this Person Group was modified. This can be null if you've never trained the group.

Figure 2: Former President Obama

- **Message:** If the training fails for any reason, you'll find the reason for the failure here. For success, this will be blank.

All there is left to do now is to call the /training URL in a loop, and when the status succeeds, ensure that you stop calling the loop. This is essential because you are under a rate limit, and you're paying for the calls you make. You don't want to call too often, and you don't want to call unnecessarily. The code for training the Person Group can be seen in **Listing 6**.

Create the Person Group

Congratulations! With your model complete, now you can send an input picture and have AI identify the Person.

For my example, I sent three pictures of three people. In **Figure 2**, are the three pictures of the first person I sent.

I called the appropriate method to create him as a Person in the Person Group called **myfriends**, and got the following output:

```
Created personId:
{
"personId":
"66ec4630-5369-4830-a20b-0f329d6eaf56"
} for person: Obama
```

Figure 3: Me. Sorry I'm not better looking, but it didn't crash the cloud.

Okay that's great! For the second candidate, I added my own pictures, as can be seen in **Figure 3**.

When I added myself as a Person into the Person Group, I got the following GUID as my identifier:

```
{"personId":
  "f530e1ee-a5b9-4a7b-9386-616b55525de6"}
```

Finally, I added Donald J. Trump as the third person to round up our stellar group of individuals. The specific pictures I added can be seen in **Figure 4**.

When I added the current president into the person group, I got the following GUID as my identifier:

Listing 6: Train the Person Group

```
export function trainPersonGroup(
   personGroupId: string): Promise<boolean> {
    const promise = new Promise<boolean>((resolve, reject) => {
        const requestOptions = getRequestOptions();
        requestOptions.headers['Content-Type'] = 'application/json';
        request.post(
            config.face.endPoint + '/persongroups/' +
            personGroupId + '/train',
            requestOptions,
            (err, response, body) => {
                if (err) { reject(false); }
                else {
                    const interval = setInterval(() => {
                        request.get(
                            config.face.endPoint +
          '/persongroups/' + personGroupId + '/training',
                            requestOptions,
                            (err, response, body) => {
                                if (JSON.parse(body).status) {
                                    clearInterval(interval);
                                    resolve(true);
                                }
                                else {
                                    console.log('Not trained:');
                                    console.log(body);
                                }
                            }
                        )
                    }, 1000);
                }
            }
        );
    });
    return promise;
}
```

Listing 7: Detect Face and return Face ID

```
export function detectFace(
  fileName: string): Promise<string> {
    const promise = new Promise<string>((resolve, reject) => {
        const requestOptions = getRequestOptions();
        requestOptions.body =
          fileHelpers.readImage(__dirname + "/" + fileName);
        const params = {
            "returnFaceId": "true",
            "returnFaceLandmarks": "false"
        };

        const uri =
          config.face.endPoint + "/detect?" +
          querystring.stringify(params);
            request.post(
                uri,
                requestOptions,
                (err, response, body) => {
                    resolve(JSON.parse(body)[0].faceId);
                }
            );
    });
    return promise;
}
```

Figure 4: Donald J. Trump, the third person in the cloud

Figure 5: The input image

```
{"personId":
  "507e5429-8513-40db-a282-ae3d33d165a4"}
```

Now that I've registered three persons with their faces, let's supply a sample image and run the identification logic.

Identify and Recognize the Person

The way recognition works is that first you have to identify the Person. If the last sentence didn't make any sense, here's what I mean. First you have to send the picture to the Face API and have the Face API detect a face in the picture. In doing so, the Face API returns a GUID of the face. This GUID is temporary in nature; it's only good for 48 hours. Remember, the Person IDs are permanent. The code to detect the face and return a Face ID can be seen in **Listing 7**.

Once you have the Face ID, you can identify the person. You do so by issuing a POST request to the following URL:

```
config.face.endPoint + '/identify'
```

Also, you include the following body:

```
{
 'personGroupId': personGroupId,
 "faceIds": [faceId],
 "maxNumOfCandidatesReturned": 1,
 "confidenceThreshold": 0.5
}
```

Note that in the body, you specify the number of candidates you'd like to have in the result set, and the minimum confidence threshold the recognition should meet. Also, you pass in the Face ID that you got from **Listing 7**.

At this point, all you have to do now is to make the request to get the Face ID, and then another request to identify the Person. The code can be seen in **Listing 8**.

My input image was as shown in **Figure 5**.

The result I got can be seen below:

```
Input recognized as:
[{"faceId":
 "1990e7b2-490e-49fc-a0bd-c25c39eb7a7f",
"candidates":[{
  "personId":
 "66ec4630-5369-4830-a20b-0f329d6eaf56",
   "confidence":0.77458}
]}]
```

If you match GUID 66ec4630-5369-4830-a20b-0f329d6eaf56 with the Person IDs above, you'll see that it's President Obama. I realize that he is a very famous person and our dataset was only three people, but this can scale to 100 million people.

Set Up the Code

This article had a lot of steps, and I wasn't able to show every single method in this article. However, you can get the full source code and set it up on your computer easily. Just follow the following steps:

1. Clone the following github repository: https://github.com/maliksahil/facerecognition.
2. Provision an instance of the Face API in Azure, which is explained earlier this article.
3. Get the endpoint and the two keys for the provisioned Face API.
4. Place the two keys and the endpoint in the config.ts file.
5. Ensure that you have Node.js version 8x or newer installed, and NPM 4 or 5 installed.
6. Run npm install.

Now, with the project open in Visual Studio Code, examine the index.ts.

You'll find code there, commented out as step #1, step #2, and step #3. To perform the steps of creating a Per-

Listing 8: Identify a Person.

```
faceHelpers.detectFace('./input.jpg').then(
  faceId => {
    faceHelpers.identifyPerson(
      personGroupId, faceId).then(result => {
        console.log('Input recognized as: ' + result);
    });
});
```

son Group, train the Person Group, and detect and identify a Person. Finally, do step #4 to delete the Person Group when you are done.

You can uncomment the relevant portions, hit F5 to perform the necessary steps one by one, and then check the debug log.

Conclusion

What's the value of being able to send in a picture, and have the computer recognize who it is in a matter of seconds, across 100 million people?

Imagine a stadium, where at the entrance people read a sign. And by that sign is a small camera, comparing every person with possible terrorists?

Or, what if you walk into a store, and a camera there instantly recognizes you and sends you discount codes on your phone that you must use in the next 20 minutes at that store.

Or perhaps something more mundane: You walk into the room, the computer recognizes you, and automatically sets the lights and music per your preference.

The possibilities are endless, and this is just a single capability of the Microsoft Cognitive Services. I hope to talk about those among other things, in future articles.

Until then, happy coding!

Sahil Malik

JavaScript Corner: Variables and Scope

In this series, I'll be sharing with you information and insights on what is one of the most popular programming languages in the world: JavaScript. Once upon a time, JavaScript was a simple client-side scripting language for the browser. Today, JavaScript exists on the server as well as the client and can be used to customize PDF files. Like any other programming language,

John V. Petersen
johnvpetersen@gmail.com
about.me/johnvpetersen
@johnvpetersen

Based near Philadelphia, Pennsylvania, John has been an information technology developer, consultant, and author for over 25 years.

JavaScript has its quirks and isn't perfect. For example, working with floating point numbers may be at odds with your need for precise numerical accuracy. No matter your vocation, to be effective, you need multiple tools in your toolbox, and that means, in a world of nails, JavaScript shouldn't be your only hammer.

> Controlling variables and their scope is necessary for robust and maintainable JavaScript code.

The column's structure will be concise, like my Simplest Thing Possible series. Each column will be divided into four parts: context, core concepts, application, and key takeaways.

- Before I can delve into the details, context is necessary. In the context, I'll set forth the issue or problem to be tackled.
- The core concepts begin the journey into the details of how to solve the issue or problem set forth in the context. The core concepts consist of the specific language features covered in each column.
- In the application section, I'll join theory to practice so that you can apply the material covered in the column.
- in the key takeaways, I offer guidance and highlight the points you should consider when applying the column's material.

Let's get started!

> In JavaScript, a block is what exists between curly braces ({}).

Context

The context for the first JavaScript Corner column is to tackle the issue of JavaScript variables and scope. Prior to ECMAScript 6/2015, you either declared variables explicitly with the var statement or implicitly by assigning a value to a previously unused variable name. Depending on the circumstances, the variable was either local or global in scope. If the variable was declared outside of a function, the variable was global in scope regardless of whether the var statement was used, meaning that it was available to and could be changed by any other code in your application. On the other hand, if a variable is declared in a function using the var statement, the variable has local scope limited to that function as well as any blocks within that function. If the var statement isn't used, regardless of the fact that the variable is declared inside a function, that variable has global scope.

Today, there are two additional statements, const and let, which help us control variables and their scope. Controlling variables and scope is necessary for robust and maintainable JavaScript code. As before, global and local scope is supported. In addition, block scoping is also supported. In the next section, I discuss the specific language features affected.

Core Concepts

The three language elements covered in this column include var, const, and let.

- **var:** The var statement declares a variable and optionally, you can assign a value at the same time. For example, the following are valid ways to declare global variables in JavaScript:

```
var x;
var y = 1;
z = 1;
var a,b,c = 1;
```

To make the statements for variables x, y, a, b, and c local, you just need to wrap them in a function as follows:

```
function declare() {
  var x; //local
  var y = 1; //local
  var a,b,c = 1; //local
  z = 1; //global
}
```

Note that the z variable will still be global because the var statement is omitted.

- **const:** New as of ECMAScript 6/2015, the const statement declares a block scope constant variable. The variable is constant to the extent that the variable cannot be re-declared nor can its value change within the same scope. If the variable is an object, property values can change.
- **let:** New as of ECMAScript 6/2015, the let statement declares a block scope variable where its value can change but can only be declared once in a block. That block scope may be global or local depending on where the variable is declared. If declared outside a function, that particular instance will have global scope. If declared in a function or a block within a function, it will have function or

block scope respectively. What if the variable is declared once at the function level and then again at a lower block level? Unlike previous JavaScript versions, while the lower block is executing, the new variable value controls. Upon exiting that block, the old value again controls.

Application

The applications that follow are used for illustration purposes only to show how these features work. They should not be taken as recommended approaches.

var

The following is a short review of the var statement's function level scope and the lack of block level scoping:

```
var x = 1;
(function() {
   var x = 2;
   console.log(x);
   //Output 2
  {
         var x = 3
   console.log(x);
   //Output 3
  }
   console.log(x);
   //Output 3 because var is only function
   //scope not block scoped
})();
console.log(x);
//Output 1
```

const

The following demonstrates that once a const has been declared and assigned (and the two must occur together), its value cannot change:

```
const x = 1;
x = 2;
//Results in a Assignment to constant
//variable error
```

The prohibition on value changes doesn't extend to object properties. The following is permissible:

```
const x = {"Name" : "x"};
console.log(x.Name);
//Output x
x.Name = 'X';
console.log(x.Name);
//Output X
```

The following illustrates that it's impermissible to redeclare a const at the same block level:

```
const x = 2;
const x = 1;
//Results in an Identifier  'x'
//as already been
declared error
```

If different blocks are involved, because const has block scope, earlier variable declarations are available to later blocks. If the same variable name is used in a later block, the earlier declaration value is preserved and reinstated after the later block executes:

```
const x = 1;
(function() {
   const x = 2;
   console.log(x);
   //Output 2
   {
   const x = 3;
   console.log(x)
   //Output 3
   }
})();
console.log(x);
//Output 1
```

let

The best way to think of the let statement is that it's like const plus the ability to re-assign values. Assuming that you can rely on full ECMAScript 6/2015 support, if you need to re-assign values, use let. If you don't need to re-assign values or the value is an object, use const.

Key Takeaways

The var, const, and let statements are only alike in that they are ways of declaring JavaScript variables.

- The var statement is only global and local/function scoped.
- The const and let statements go a step further to supporting block scoping. A const variable within the same scope can only be declared once and its value cannot change. If the value is an object, individual properties can change at any time.
- The let statement is like const with respect to scoping and, like const, a let variable cannot be redeclared within the same scope. You can reassign values using let.

How const and let function is straightforward both as to how they are different from each other, how they differ from the var statement and how their scope can be managed. The main question you may have right now is whether you should stop using the var statement in favor of the let and const statements.

For existing code, assuming you have used good practices like the module-pattern, there's no reason to ever change that code. For new code, in my opinion, it's too early to discard var entirely. If you do choose to go that route, be sure to thoroughly test your code against the browser you support.

Alternatively, consider using a transpiler like Traceur that allows you to write tomorrow's JavaScript and take advantage of those features today and have it automatically converted to today's JavaScript that you know will work across most/all browsers. Transpilers are a great way to manage present development with the latest language version in the context varying support for new language features. A good resource to determine browser support is here: http://kangax.github.io/compat-table/es6/.

John V. Petersen

CODE

JavaScript Language Reference

In my opinion, the best, most accurate, and comprehensive online language reference is hosted by Mozilla: https://developer.mozilla.org/en-US/docs/Web/JavaScript. You should absolutely have this site bookmarked in your favorite browser.

Refactoring a Reporting Services Report with Some SQL Magic

I've been insanely busy the last several months. I had to put together a few prototypes outside of my normal work schedule (translation: I had to do it after midnight). The prototypes worked well enough for the users, but they didn't work well under the hood. I remember an old quote from Borland's founder, Philippe Kahn. I forget his exact words, but they were something close to, "First you make it work; then you make it work well." But sometimes the question is, "When is 'then'?" Let's all confess that we accumulate some developer debt, so the key is to clean the slate as quickly as possible. Sometimes it's just a matter of going off into a quiet room and restating the initial objective. That's my story for this issue.

Kevin S. Goff
kgoff@kevinsgoff.net
http://www.KevinSGoff.net
@KevinSGoff

Kevin S. Goff is a Microsoft Data Platform MVP. He has been a Microsoft MVP since 2005. He is a database architect/developer/speaker/author, and has been writing for CODE Magazine since 2004. He's a frequent speaker at community events in the Mid-Atlantic region and also spoke regularly for the VS Live/Live 360 Conference brand from 2012 through 2015. He creates custom webcasts on SQL/BI topics on his website.

A Mistake in 1987, a Mistake in 2018

When I was 22 years old, I tried to write a multi-column picklist utility using Turbo-C. It "worked," and that's the best thing I'll say about it. I wrote about ten times more code than I needed for the utility and the code was nasty. Of course, I was young and needed guidance. Fortunately, another developer helped me to simplify everything and the experience made me wiser.

Well, there's nothing like some self-service humble pie to remind us how easily we can fall back into old habits. Recently, I had to create a dashboard page in SQL Server Reporting Services with X number of pie charts, similar to the basic example in **Figure 1**.

> Contrary to popular belief, hard-wiring values in code won't kill kittens, but it does create countless irritations down the road.

Figure 1 shows an output page of six pie charts, one for each country. Each country's pie chart breaks out sales by product.

OK, so what's the problem? Well, ask the developer (me) what happens if we have sales for a seventh country. If the developer's only interest is the quickest and most expedient fix, the developer does the following:

- Open up the SSRS designer for the report (**Figure 2**), which shows six report objects.
- Add a seventh report chart object by copying and pasting, much in the same way he copied/pasted to create the first six charts. (You can probably see where this example is going).
- Filter that seventh chart object on the seventh country, just like he did when he hard-wired country names in the chart filter for each of the first six chart objects (**Figure 3**).
- Say a prayer that no one else on the developer team will ever see this foolishness!

Yes, the ugly truth here is that this relatively attractive output back in **Figure 1** has two nasty features under the hood: hard-wired countries and a dependency on a new chart object for each country. As far as developer sins go, my soul better start praying!

There's an old developer joke that when you do something very bad, it will "kill kittens." Well, hard-wiring values in code won't kill kittens. But this kind of developer debt creates countless irritations (or worse) down the road. If you change the attributes for one chart, you have to manually change all six charts, if you remember to do so. The mere fact that you'd have to remember repetitive acts is proof enough that you have a very bad anti-pattern here. Therefore, developers should be tortured for hard-wiring values without first putting in the effort to create a more elegant solution!

OK, so hard-wiring country values and creating a new chart object for a new country is almost like begging for termination. However, when you're using a reporting tool like SSRS, which is neither object-oriented nor fully programmable, how can you create a more generic solution?

I went back to the original problem. I wanted to create a solution that would show pie charts for three countries across the top, and as many rows as necessary. Essentially, I wanted one massive matrix with charts inside the...**wait a minute, did I say matrix**? SSRS contains a matrix report object! But how can I define it so (for instance) the first three countries occupy the first row and columns 1–3, the next three countries would occupy the second row and columns 1–3, etc.

Well, I've (mostly) trained my mind that the SQL RANK functions work well for generating sequences of numbers. Therefore, I could use the SQL ranking functions to assign row and column numbers for each country (in alphabetic order), and then use those row/column numbers as the row/column group values in the matrix! Sometimes it's amazing how you can start to see a solution just by talking out the problem and thinking about prior software functions you've used to solve problems. (Some academics say this act is the basis of composition).

RANK Functions to the Rescue!!!

To repeat my goal, I want to create a matrix where each chart occupies a cell and that chart plots sales for the products in a specific country. I want to generate specific sequential row/column values that I can supply as row/

Beauty is Only "UI" Deep

An attractive visualization might have an equally elegant engine under the hood—or not. I won't debate whether it's more important to first make something look nice. I will debate if you get complacent about the beauty inside.

group properties to the matrix (the result set in **Figure 4**), where each row/group combination represents a pie chart for a country.

Here's where I can use the SQL ranking functions. If I assign a rank number for each country, I can define row/column values based on the assumption of three columns (charts) per row.

Let's take Australia, Canada, and France, which have DENSE_RANK values of 1 through 3 based on name order. If I subtract 1 from each dense_rank, divide by 3, and then add 1 to the result, I'll have a value of 1 that I can use to indicate that they'll occupy row 1.

Similarly, if I take the same dense_rank, subtract 1, and then take the remainder after dividing by 3 (and

Figure 1: A basic SSRS dashboard page with a pie chart for each country

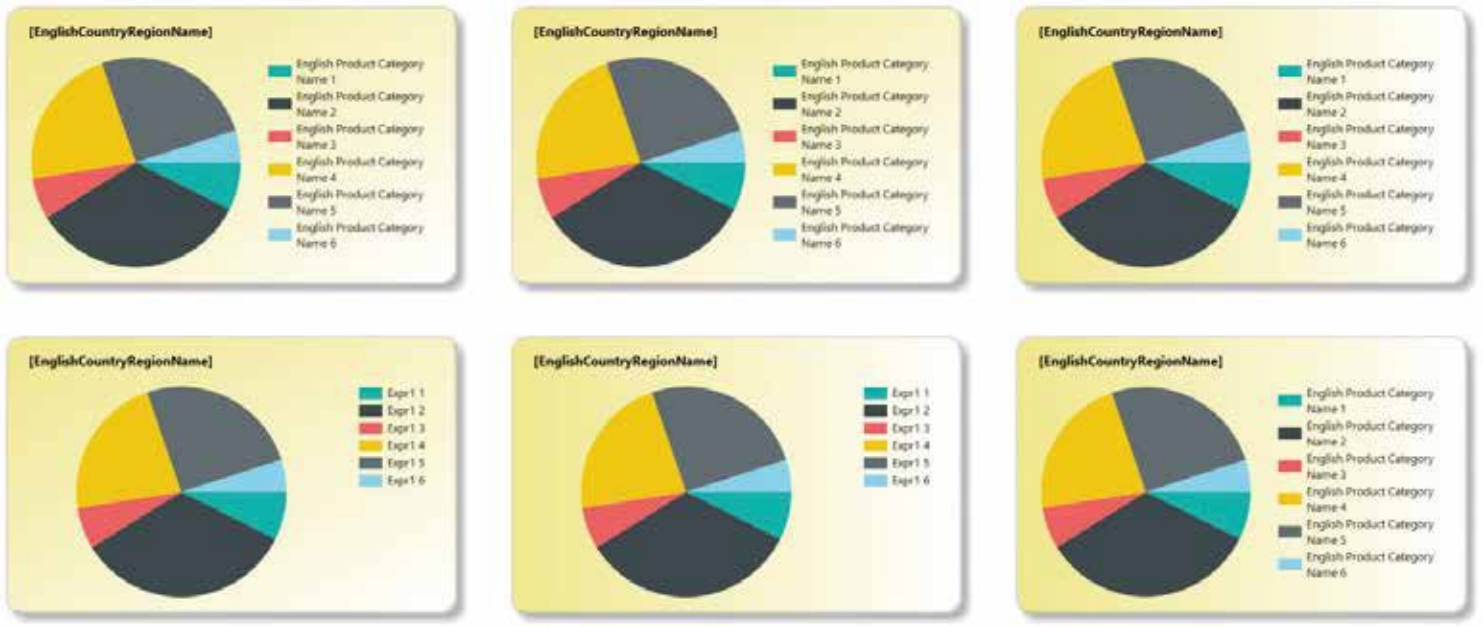


Figure 2: The SSRS designer with six chart objects, one for each country

then adding 1), I'll have the values of 1 through 3 that I can use to indicate that they'll occupy columns 1 through 3.

```
SELECT
((DenseRankNum-1) / 3) + 1 as RowNum,
(( DenseRankNum  -1) % 3) + 1 as ColNum,* FROM
    (SELECT DENSE_RANK() OVER
        ( ORDER BY englishcountryregionname)
            as DenseRankNum, …
```

Listing 1 contains the full query for the solution. Now that I have some supplemental values in the result set, I can do the following (as seen in **Figure 5**):

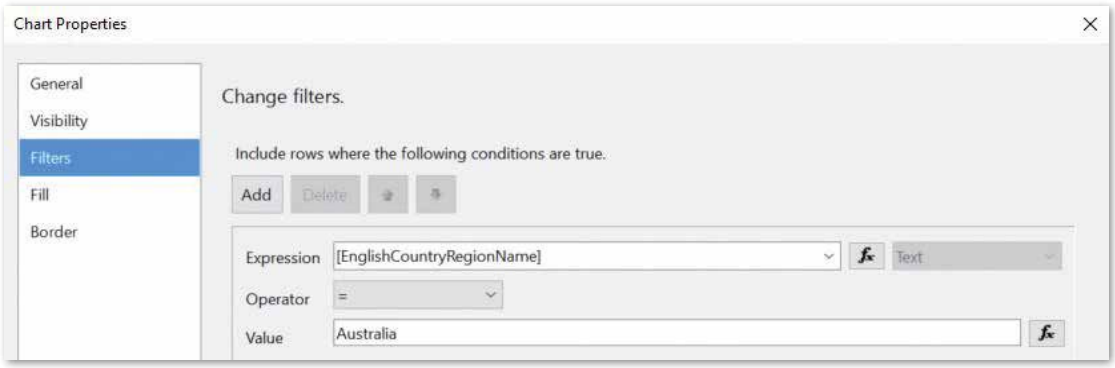


Figure 3: The chart properties for each of the six charts with a hard-wired filter for each country.

	RowNum	ColNum	DenseRankNum	EnglishCountryRegionName	EnglishProductCategoryName	SalesAmount
1	1	1	1	Australia	Accessories	1293.0361
2	1	1	1	Australia	Bikes	17616.37
3	1	1	1	Australia	Clothing	6725.5533
4	1	1	1	Australia	Components	6847.4693
5	1	2	2	Canada	Accessories	18377.4742
6	1	2	2	Canada	Bikes	405536.2603
7	1	2	2	Canada	Clothing	54626.8764
8	1	2	2	Canada	Components	45386.9296
9	1	3	3	France	Accessories	10616.1223
10	1	3	3	France	Bikes	239306.9953
11	1	3	3	France	Clothing	17502.0116
12	1	3	3	France	Components	18465.8012
13	2	1	4	Germany	Accessories	6020.7996
14	2	1	4	Germany	Bikes	90833.3098
15	2	1	4	Germany	Clothing	10197.2951
16	2	2	5	United Kingdom	Accessories	7506.4325
17	2	2	5	United Kingdom	Bikes	163178.5812
18	2	2	5	United Kingdom	Clothing	16923.5811
19	2	2	5	United Kingdom	Components	34574.712
20	2	3	6	United States	Accessories	37874.0761
21	2	3	6	United States	Bikes	1499265.51…
22	2	3	6	United States	Clothing	126169.6182
23	2	3	6	United States	Components	56376.8619

Figure 4: The result set for a generic chart solution, with generated row/column values

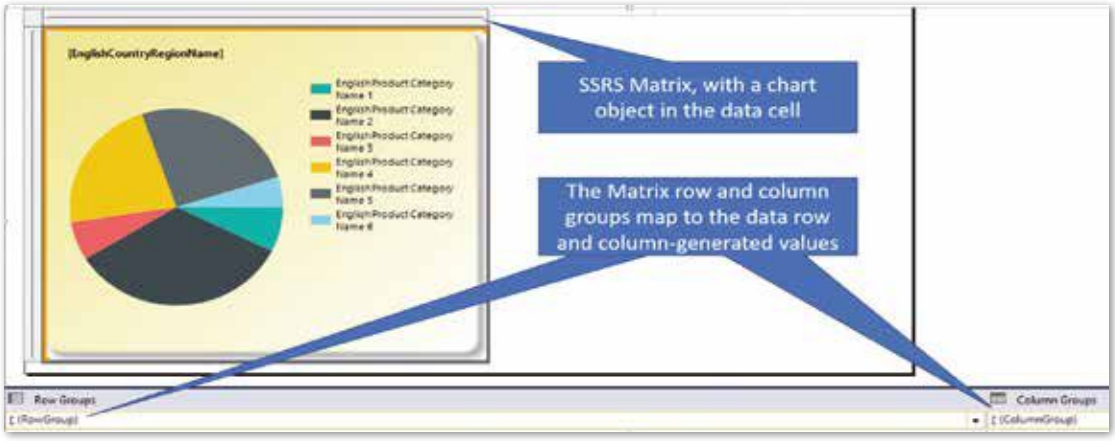


Figure 5: One chart inside a matrix in SSRS, with row/column groups mapped to the values

Know Your Software Tools and Read How Others Use Them

I've read many online articles on the SSRS matrix that leave me humbled. I'd rather be humbled than uninformed. No matter how much you know about a software or language feature, there's always someone who has successfully done something you never would have thought to try.

Listing 1: Full SQL code for reporting solution

```
SELECT
((DenseRankNum-1) / 3) + 1 as RowNum,
(( DenseRankNum  -1) % 3) + 1 as ColNum,* FROM
      (SELECT DENSE_RANK() OVER
          (  order by   englishcountryregionname) as DenseRankNum,
    DimGeography.EnglishCountryRegionName,
    DimProductCategory.EnglishProductCategoryName,
    SUM(FactResellerSales.SalesAmount) AS SalesAmount
    FROM   FactResellerSales
       INNER JOIN DimReseller
          ON FactResellerSales.ResellerKey = DimReseller.ResellerKey
       INNER JOIN DimGeography
          ON DimReseller.GeographyKey = DimGeography.GeographyKey
          INNER JOIN DimProduct
             ON DimProduct.ProductKey = FactResellerSales.ProductKey
          INNER JOIN DimProductSubcategory
             ON DimProductSubcategory.ProductSubcategoryKey =
                DimProduct.ProductSubcategoryKey
           INNER JOIN DimProductCategory
              ON DimProductCategory.ProductCategoryKey =
                 DimProductSubcategory.ProductCategoryKey
       WHERE (FactResellerSales.PromotionKey = 2)
       GROUP BY DimGeography.EnglishCountryRegionName,
           DimProductCategory.EnglishProductCategoryName ) t
ORDER BY EnglishCountryRegionName, EnglishProductCategoryName
```

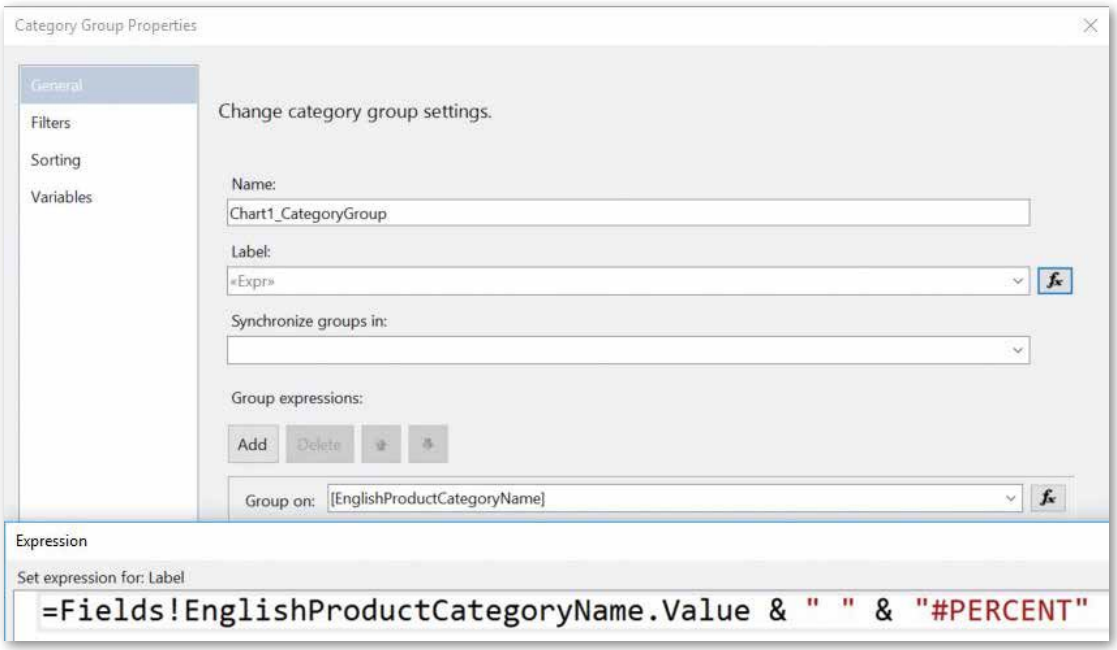


Figure 6: You can define a label in the chart category group to show each product's percent of total

- Add an SSRS matrix control to the SSRS report.
- Map the matrix row and column group properties to the new rownum/colnum values in the result set.
- Add the pie chart as the lone data cell in the matrix (and stretch horizontally and vertically as needed).

Quite often, creating a generic and data-driven solution simply means creating some semi-intelligent data for software controls to use. You can build many solutions using the SSRS matrix, especially when you supplement the matrix with control values as I've done here.

> You can build many solutions using the SSRS Matrix, especially when you supplement the matrix with control values.

Also, if you're wondering how I was able to add the percentage of total in the legend text (which I personally prefer over data label lines), you can define a custom label in the chart's category group properties (**Figure 6**).

Doctors and Developers

I sometimes compare the medical industry to the software industry, with analogies ranging from serious to tongue-in-cheek. When you build a solution like the one I initially built (hard-wired values and propagated objects via copy-and-paste) with no serious intention to improve it, you're essentially putting a sick patient back on the street. Yes, that patient might walk the street with no visible signs of breaking down, but eventually that patient will collapse.

We've all created solutions we're not proud of, ones where design reviewers would rip it apart. Don't fret. Be your own critic. Take action and improve where you can. If you know you have five sick patients walking the street, make a plan to cure those patients. Get busy!!!

Kevin S. Goff

Everything Is a Journey, So Keep Moving

I used this line in my last article. The line has tremendous personal meaning for me, so I'll use it again. Exactly one year ago this week, I learned that I had serious health issues. My doctor gave me some "tough love" but also talked about goals as journeys. I dropped 60 pounds and got healthy. Still, the journey doesn't end. When a team member accomplishes anything, I tell the person, "Good work… keep at it…keep the motor running.'

Understanding Blockchain: A Beginner's Guide to Ethereum Smart Contract Programming

One of the hottest technologies of late is blockchain. A blockchain is a digital transaction of records that's arranged in chunks of data called blocks. These blocks link with one another through a cryptographic validation known as a hashing function. Linked together, these blocks form an unbroken chain—a blockchain. A blockchain is programmed to record

Wei-Meng Lee
weimenglee@learn2develop.net
www.learn2develop.net
@weimenglee

Wei-Meng Lee is a technologist and founder of Developer Learning Solutions (http://www.learn2develop.net), a technology company specializing in hands-on training on the latest technologies. Wei-Meng has many years of training experiences and his training courses place special emphasis on the learning-by-doing approach. His hands-on approach to learning programming makes understanding the subject much easier than reading books, tutorials, and documentation. His name regularly appears in online and print publications such as DevX.com, MobiForge.com, and CODE Magazine.

not only financial transactions but virtually everything of value. Another name for blockchain is distributed ledger.

In this article, I explain the basic ideas behind a blockchain and how it works. Once the fundamentals of blockchain are out of the way, I dive into one of the key features behind the Ethereum blockchain: *smart contracts*. Smart contracts allow developers to deploy decentralized applications that take advantage of the various characteristics of blockchain. Hold on tight, as I'm going to discuss a lot of concepts in this article. But if you follow along closely, you'll understand the concepts of blockchain and be on your way to creating some really creative applications using it!

Centralization vs. Decentralization

In the traditional client-server architecture, you store transactions records in a database in a centralized location. **Figure 1** shows the interactions between clients and servers.

Storing your data in a central location has the following risks:

- The potential for data loss
- The potential for illegal data alteration

The first point is easy to mitigate and can be solved by replicating the data in multiple locations (backups). The second point is more challenging. How do you ensure the integrity of the data stored on the server? A good example is banks. How can you be sure that the balance in your bank account reflects the correct amount of money you possess? In most countries, people trust their banks and governments to maintain the correct records of their personal possessions. But in some countries, governments are corrupt and people have very little trust in them. As you can see, a centralized approach to data storage isn't always ideal.

Thus, **blockchain** was born, focusing on the decentralization of data storage, commonly known as a **distributed ledger**. Using decentralization, **Figure 1** now looks like **Figure 2**.

Storing the transactions on multiple computers ensures that no single computer can singlehandedly alter the data on its own, because the transactions are replicated on multiple computers. If a malicious actor wishes to alter the transactions, he must modify the transaction not only on a single computer, but also on all the computers holding the transactions. The more computers participating in the network, the more computers he needs to modify. In this case, decentralization shifts the trust from a central authority to one that is trustless: You don't need to trust a central authority because now everyone holds the records.

The Blocks in Blockchain

In **Figure 1** and **Figure 2**, you saw that the database contains transactions. Typical transactions may look like this:

- A sends 5 BTC (bitcoins) to B
- B sends 2 BTC to C
- A sends 1 BTC to D

It's important to note that blockchains are used not only for cryptocurrencies like bitcoins and ethers, but can be used for anything of value. Transactions are then grouped into blocks (see **Figure 3**).

Transactions are grouped into blocks so that they can be efficiently verified and then synchronized with other computers on the network. Once a block is verified, they are added to the previous block, as shown in **Figure 4**.

Blockchain gets its name from the fact that blocks of data are chained to each other cryptographically. In order to ensure the correct order of transactions in the blockchain, each block contains the hash (a hash is the result of mapping a block of data into data of fixed size using a cryptographic function) of the previous block, as shown in **Figure 5**.

Storing the hash of the previous block in the current block assures the integrity of the transactions in the previous block. Any modifications to the transaction(s) within a block causes the hash in the next block to be invalidated, and it also affects the subsequent blocks in the blockchain. If a hacker wants to modify a transaction, not only must he modify the transaction in a block, but all other subsequent blocks in the blockchain. In addition, he needs to synchronize the changes to all other computers on the network, which is a computationally expensive task to do. Data stored in the blockchain is immutable, for it's hard to change once the block they are in is added to the blockchain.

The first block in a blockchain is known as the **genesis block**. Every blockchain has its own genesis block; the bitcoin network has its own genesis block, and Ethereum has its own genesis block.

Nodes in a Blockchain Network

I have earlier mentioned that in a decentralized network, there are many computers holding onto the transactions. I can now replace the transactions with the blockchain, as shown in **Figure 6**.

Computers storing the entire blockchain are known as **full nodes**. They help to verify and relay transactions and blocks to other nodes. They also make the network robust, as there are now multiple nodes in the network with little risk of a single point of failure. Besides full nodes, there are also nodes known as **light nodes**, which I'll discuss later in this article.

When a miner has successfully mined a block, he earns mining fees as well as transaction fees. That's what keeps miners motivated to invest in mining rigs and keep them running 24/7, thereby incurring substantial electricity bills.

Miners

Among all the full nodes in a blockchain network, some are known as **mining nodes** (also known as **miners**). Miners add blocks to the blockchain. In order to add a block to the blockchain, a miner needs to do the following:

- Take the transactions in the previous block and combine it with the hash of the previous block to derive its hash.
- Store the derived hash into the current block

Figure 7 outlines the process:

The process of performing hashing is straightforward and a computer can perform that in a matter of milliseconds. So how do you ensure that all the miners have equal chances to mine a block? It turns out that to solve this problem, the blockchain network (such as Bitcoin or Ethereum) inserts a **network difficulty target** into every block, so that in order to mine a block, the result of the hash must meet the criteria set by the difficulty target. For example, a **difficulty target** may dictate that the resultant hash starts with five zeros; if not the block can't be accepted. As more miners join the network, the network automatically adjusts the difficulty target so that blocks can be mined at a constant rate.

In order to meet the difficulty target, miners need to inject a number called **nonce** (number used once) into the block. So instead of deriving a hash from the transactions and the hash of the previous block, they now add the nonce to the hashing operation. The miners need to compete with each other to guess the value of the nonce that gives a resultant hash matching the difficulty target. And that's basically all that miners do! Their job is to find the value of this nonce.

The updated blockchain now looks like **Figure 8**.

The process of finding the nonce is called **Proof-of-Work** (PoW). Once the nonce is found, the entire block and the nonce is broadcasted to other nodes, informing them that the block has been mined and is ready to be added to the blockchain. The other blocks can now verify that the nonce does indeed satisfy the difficulty target and stop their cur-

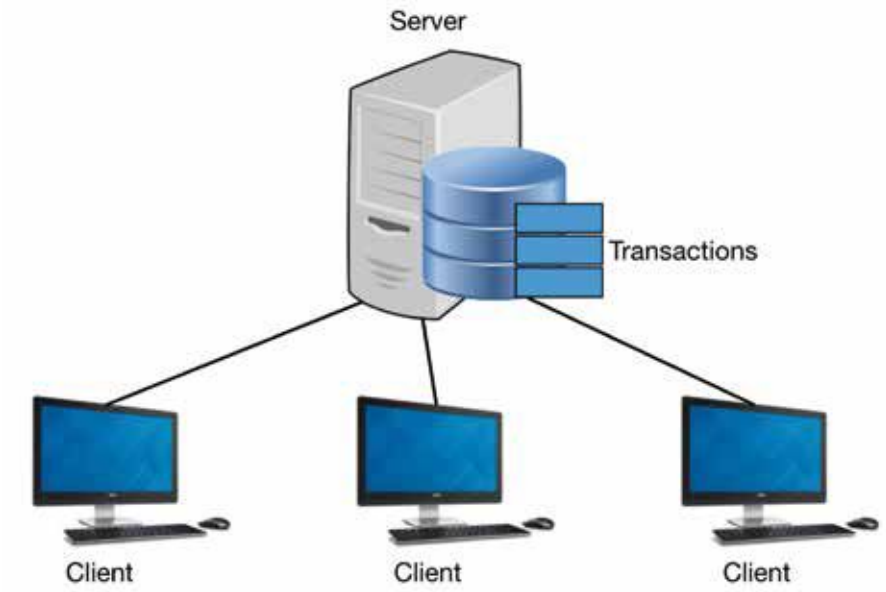


Figure 1: Centralized data storage

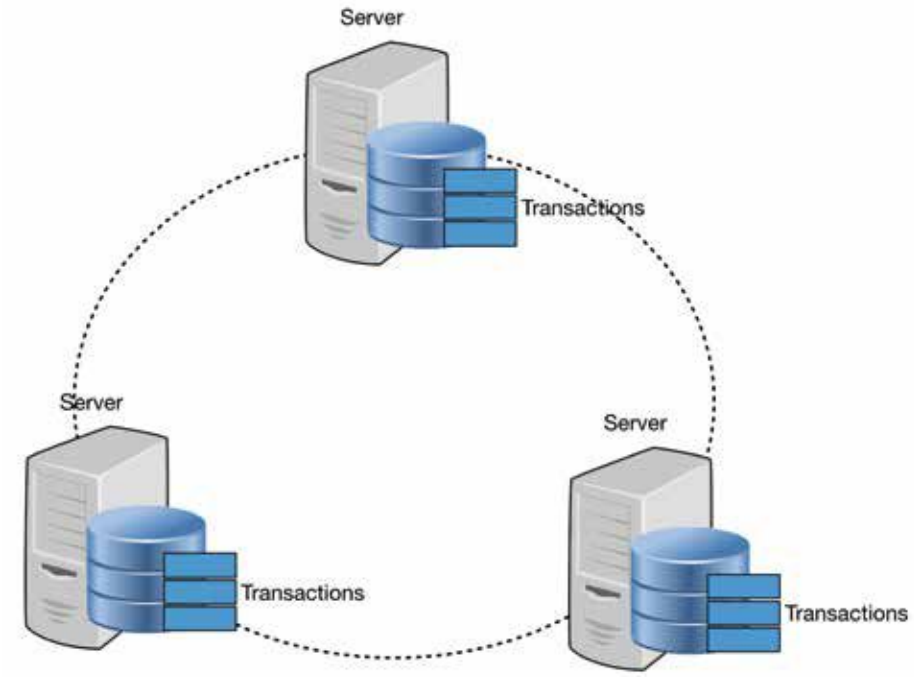


Figure 2: Decentralized data storage

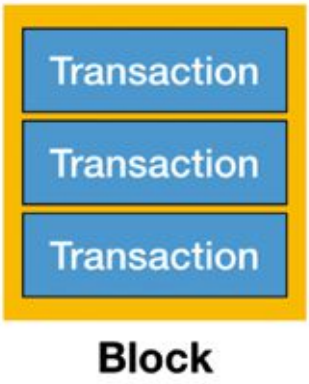


Figure 3: Transactions are grouped into blocks

What is Ethereum?

Ethereum is an open-source public blockchain that is similar to the Bitcoin network. Besides offering a cryptocurrency known as Ether (which is similar to bitcoin), the main difference between Bitcoin and Ethereum is that the latter offers a programming platform on top of the blockchain, called Smart Contract.

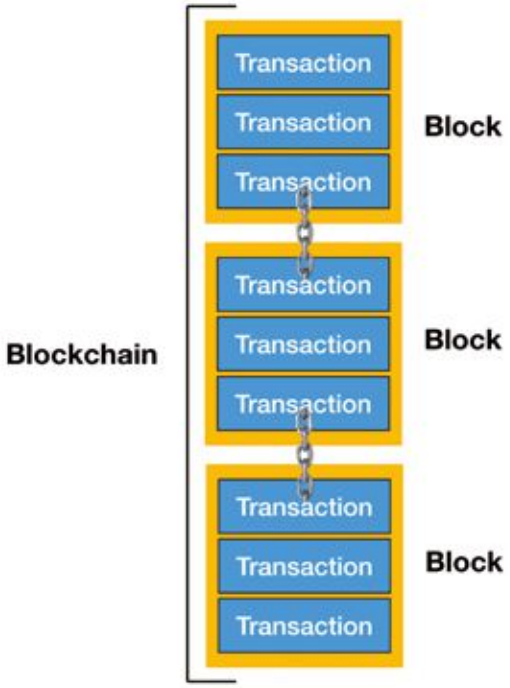


Figure 4: Linking blocks to form a blockchain

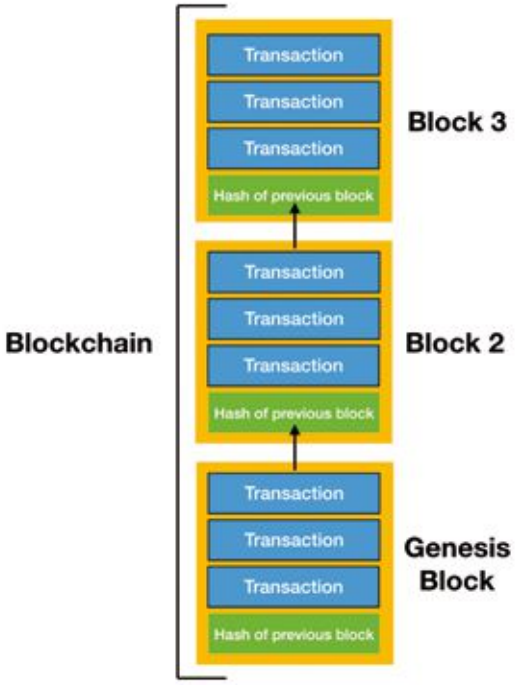


Figure 5: Using hashing to chain the blocks in a blockchain

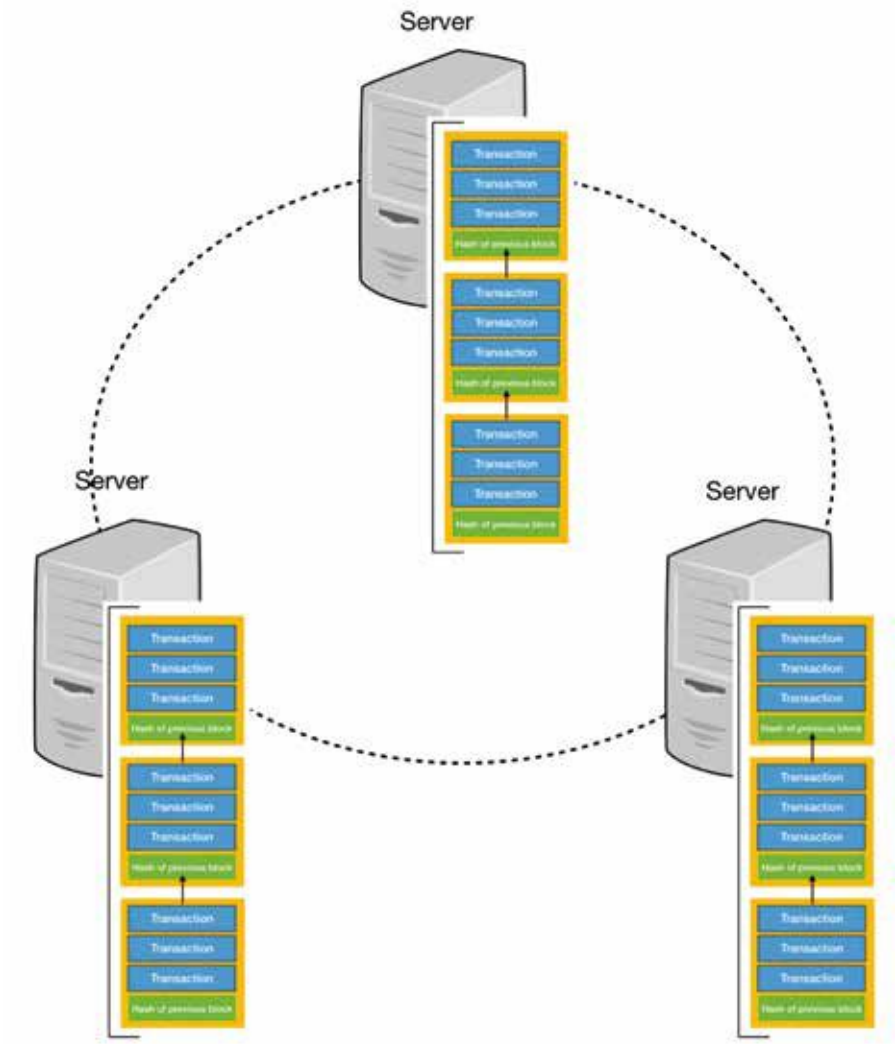


Figure 6: Full nodes in the blockchain network containing the blockchain

rent mining process and move on to mine the next block. The key principle behind PoW is that it's difficult to find the nonce but easy for others to verify once you've found it. A good analogy is a digital lock; it's difficult to find the correct key combination to unlock it but very easy to verify it once you've found the correct key combination.

Immutability of Blockchains

In a blockchain, each block is chained to its previous block through the use of a cryptographic hash. A block's identity changes if the parent's identity changes. This, in turn, causes the current block's children to change, which affects the grandchildren, and so on. A change to a block forces a recalculation of all subsequent blocks, which requires enormous computation power. This makes the blockchain immutable, a key feature of cryptocurrencies like Bitcoin and Ethereum.

> In general, once a block has six or more confirmations, it's deemed infeasible for it to be reversed. Therefore, the data stored in the blockchain is immutable.

As a new block is added to the blockchain, the block of transactions is said to be **confirmed** by the blockchain. When a block is newly added, it's deemed to have one confirmation. As another block is added to it, its number of confirmations increases. **Figure 9** shows the number of confirmations that the blocks in a blockchain has. The more confirmations a block has, the more difficult it is to remove it from the blockchain.

InterDrone

The Largest Commercial Drone Event in North America

September 5-7, Rio Hotel, Las Vegas

Join thousands of UAV professionals at the largest and most comprehensive drone conference and exposition in the United States. Over the course of 3 days, you'll learn from the best and brightest and meet face-to-face with innovative hardware and software vendors reshaping the industry.

160+ EXHIBITORS

115+ SESSIONS

100+ SPEAKERS

ENTERPRISE TRACKS

- Agriculture
- Construction
- Surveying and Mapping
- Public Safety
- Energy Inspection
- Infrastructure Inspection

Visit us at interdrone.com

Blockchain in More Detail

I mentioned that a block contains a list of transactions, the hash of the previous block, and the nonce. That was an over-simplification. In actual fact, a block also contains (see also **Figure 10**):

- A block header that includes the nonce, hash of the previous blocks, as well as the Merkle Root of the transactions (discussed in the next section)
- The list of transactions

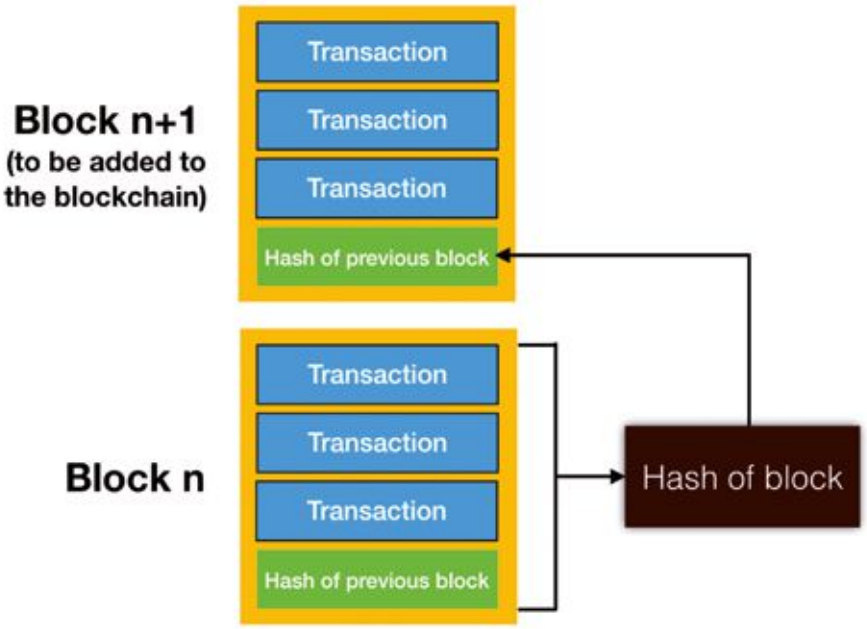


Figure 7: Storing the hash of the current block in the next block

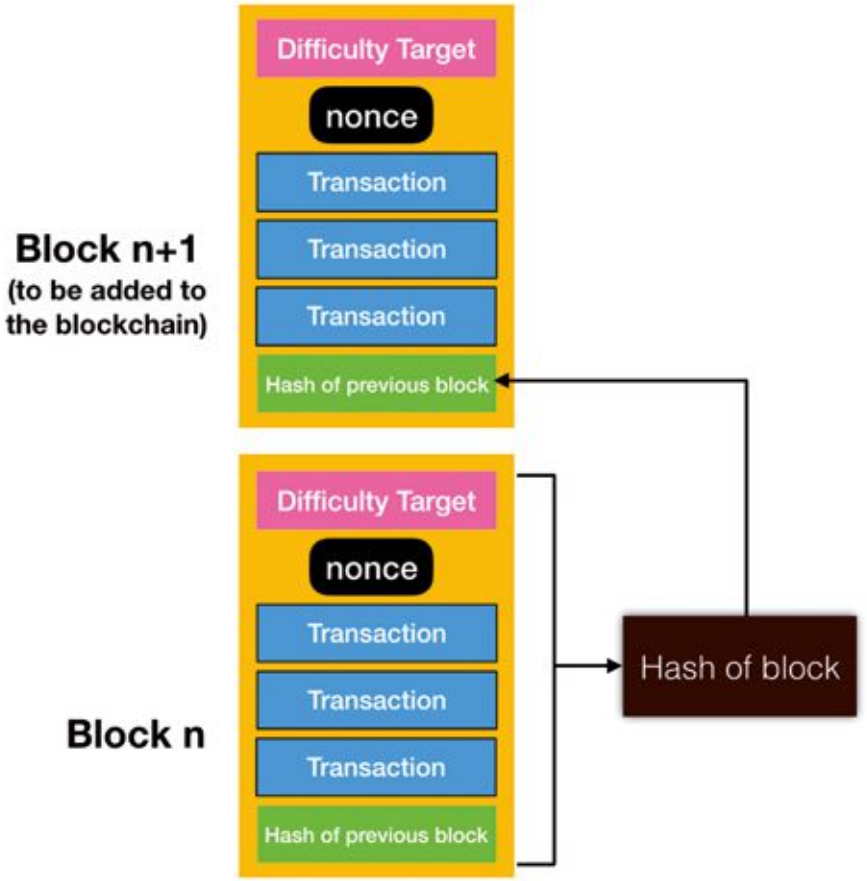


Figure 8: Miners work hard to find the value of the nonce.

Merkle Tree and Merkle Root

The list of transactions is stored as a **Merkle tree**. A Merkle tree is a tree data structure in which every leaf node is the hash of a transaction and every non-leaf node is the cryptographic hash of the child nodes. **Figure 11** shows how the **Merkle Root** is derived from the transactions.

As you can see from the figure, each transaction is hashed. The hash of each transaction is hashed together with the hash of another node. For example, the hash of transaction A (H_A) is combined with the hash of transaction B (H_B) and hashed to derive H_{AB}. This process is repeated until there's only one resultant hash. This final hash is known as the Merkle Root. In the above example, because H_E doesn't have another node to pair with, it's hashed with itself. The same applies to H_{EE}.

The Merkle Root is stored in the Block Header and the rest of the transactions are stored in the block as a Merkle tree. In the earlier discussion, I mentioned about full nodes. Full nodes download the entire blockchain, and there's another type of node (known as ***light nodes***) that downloads only the blockchain headers. Because light nodes don't download the entire blockchain, they're easier to maintain and run. Using a method called **Simplified Payment Verifications** (SPV), a light node can query a full node to verify a transaction. Examples of light nodes are cryptographic wallets.

Uses of Merkle Trees and the Merkle Root

By storing the Merkle Root in the Block Header and the transactions as a Merkle Tree in the block, a light node can easily verify if a transaction belongs to a particular block. This is how it works. Suppose a light node wants to verify that **transaction C** exists in a particular block:

- The light node queries a full node for the following hashes: H_D, H_{AB}, and H_{EEEE} (see **Figure 12**).
- Because the light node can compute H_C, it can then compute H_{CD} with H_D supplied.
- With H_{AB} supplied, it can now compute H_{ABCD}.
- With H_{EEEE} supplied, it can now compute $H_{ABCDEEEE}$ (which is the Merkle Root).
- Because the light node has the Merkle Root of the block, it can now check to see if the two Merkle Roots matches. If they match, the transaction is verified.

As you can see from this simple example, to verify a single transaction out of five transactions, only three hashes need to be retrieved from the full node. Mathematically, for n transactions in a block, it takes $\log_2 n$ hashes to verify that a transaction is in a block. For example, if there are 1024 transactions in a block, a light node only needs to request 10 hashes to verify the existence of a transaction in the block.

Smart Contracts

Although the initial use of blockchain was for cryptocurrency such as Bitcoin, blockchain offers much more than

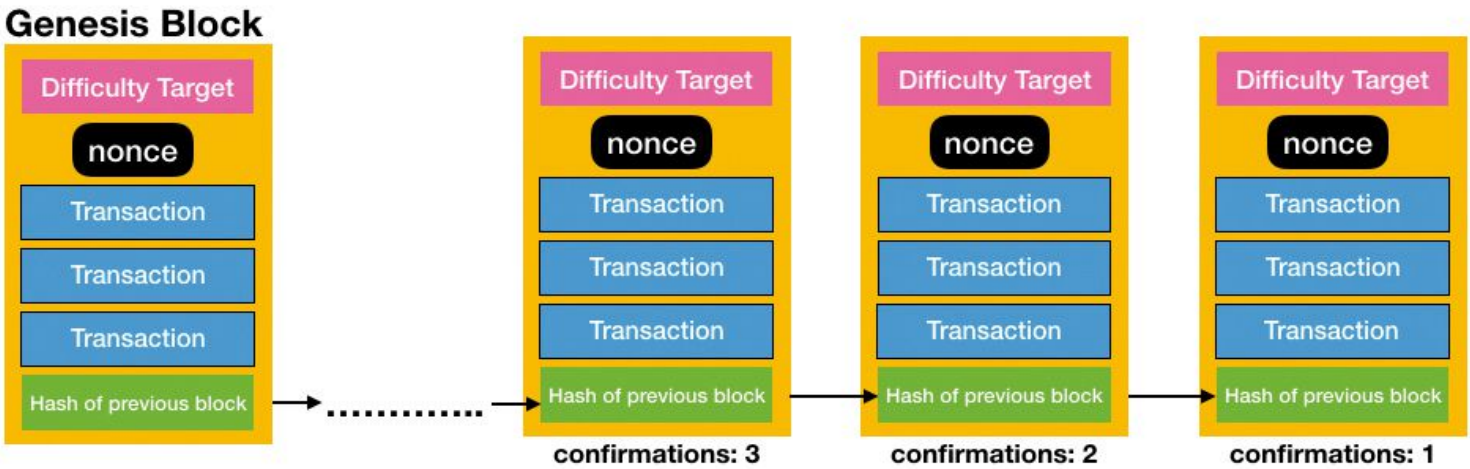


Figure 9: Confirmations of blocks in a blockchain

just a record of transactions. In addition to trading cryptocurrencies, some implementations of blockchains offer the idea of smart contracts. In particular, the Ethereum is one such example. Like Bitcoin, Ethereum offers a cryptocurrency known as Ether, but it also turned all the nodes in the network into "Turing complete" computers. What this means is that you can write programs (known as **smart contracts**) and execute them on all the nodes on the network.

Ethereum implements an execution environment on the blockchain called the **Ethereum Virtual Machine** (**EVM**). Every node participating in the network runs the EVM as part of the block verification protocol. They go through the transactions listed in the block they're verifying and run the code as triggered by the transaction within the EVM. Each and every full node in the network does the same calculations and stores the same values.

To understand how a smart contract is useful, let's imagine the following scenario. You're a musician and you want to protect your intellectual property, in this case, your music creations. You want to make sure that the lyrics of your songs are protected and that no one else can plagiarize them (especially before they are released). Because data stored on the blockchain is immutable and time-stamped, it's a good platform to store the lyrics of your songs as the proof that you're the original creator of the song. However, blockchain data are inherently public, so storing the lyrics of your creation on the blockchain isn't practical. A good workaround for this case is to store the hash of the song's lyrics. That way, you maintain confidentiality of your creation, and at the same time you can prove that the lyrics are written by you if you are able to provide the original lyrics to generate the original hash.

Based on this scenario, let's now create a smart contract that solves the problem. In Ethereum, smart contracts are written using the **Solidity** language, a language inspired by the JavaScript programming language.

The smart contract is shown in **Listing 1**. Let's examine the contract in more detail. The first statement specifies the pragma directive:

```
pragma solidity ^0.4.17;
```

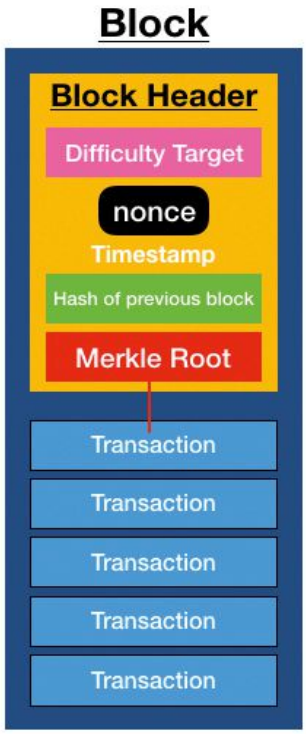


Figure 10: A block contains the block header, which in turns contains the Merkle Root of the transactions

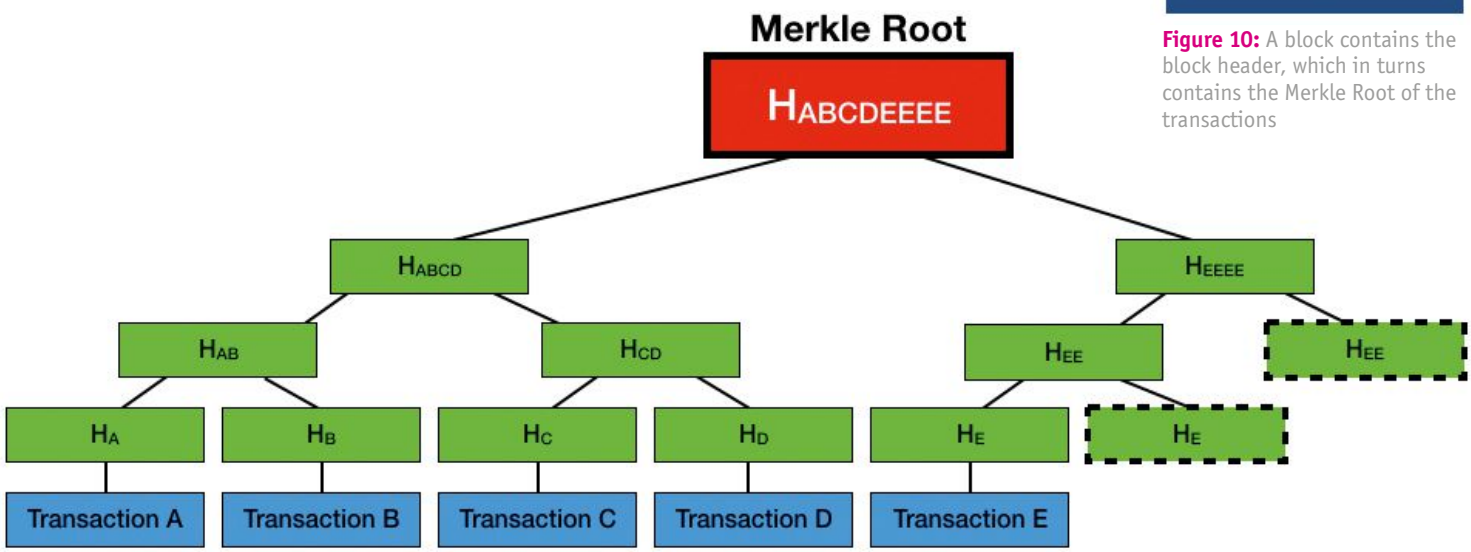
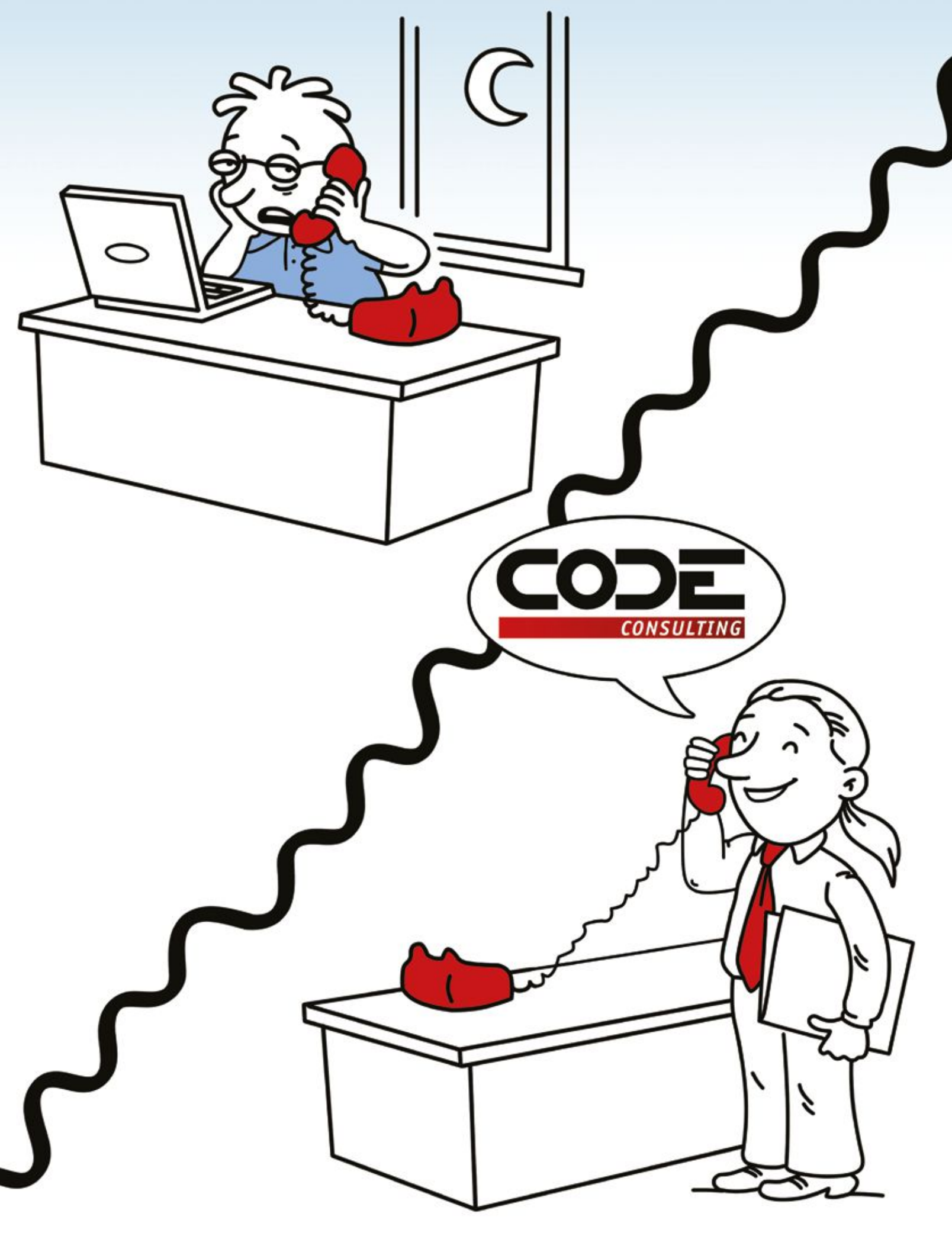
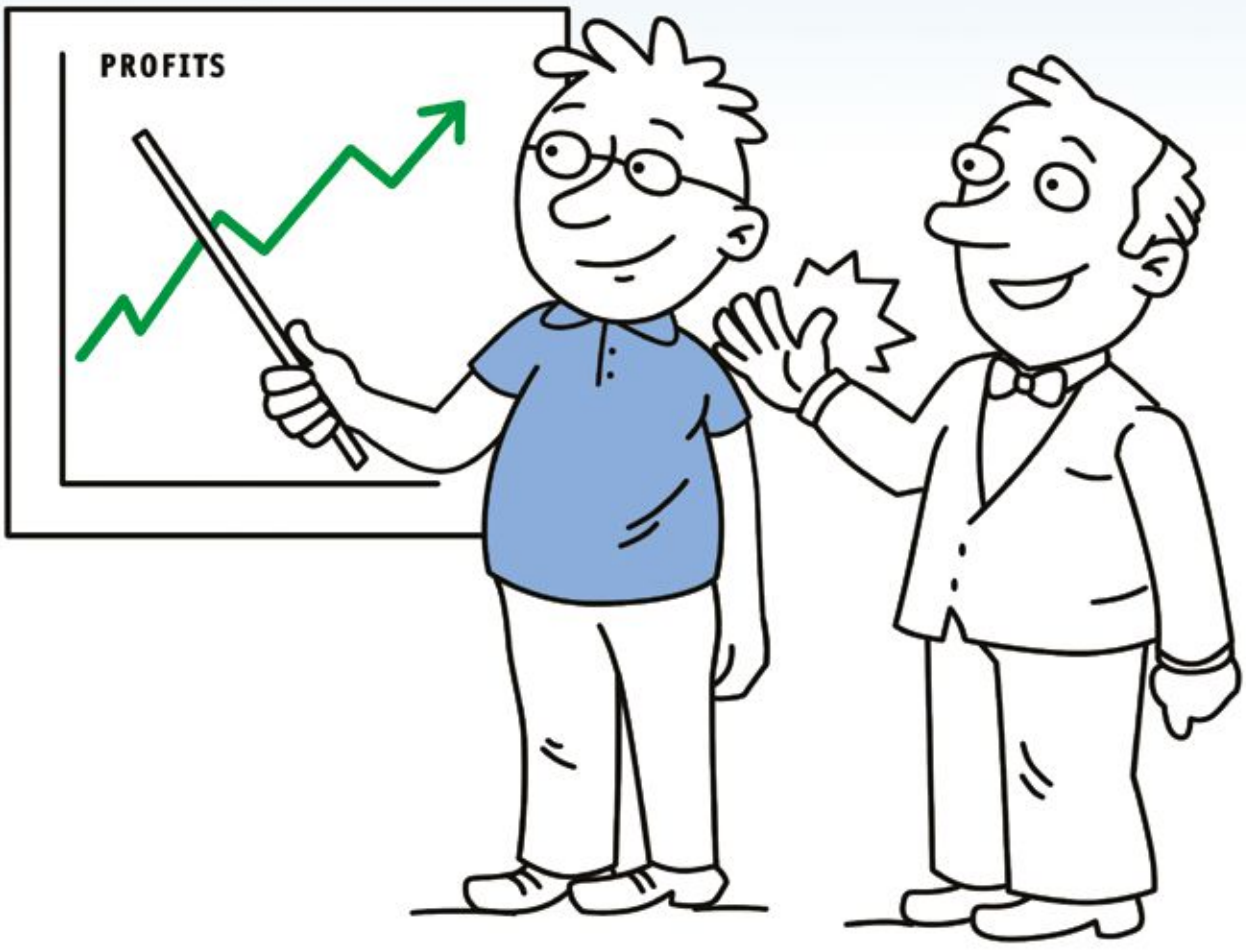


Figure 11: How the Merkle Root is derived from the Merkle Tree


CODE
CONSULTING

CODE Consulting Will Make You a Hero!



Lacking technical knowledge or the resources to keep your development project moving forward? Pulling too many all-nighters to meet impending deadlines? CODE Consulting has top-tier programmers available to fill in the technical and manpower gaps to make your team successful! With in-depth experience in .NET, Web, Azure, Mobile and more, CODE Consulting can get your software project back on track.

Contact us today for a free 1-hour consultation to see how we can help you succeed.

Helping Companies Build Better Software Since 1993

www.codemag.com/techhelp
832-717-4445 ext. 9 ● info@codemag.com

CODE CONSULTING

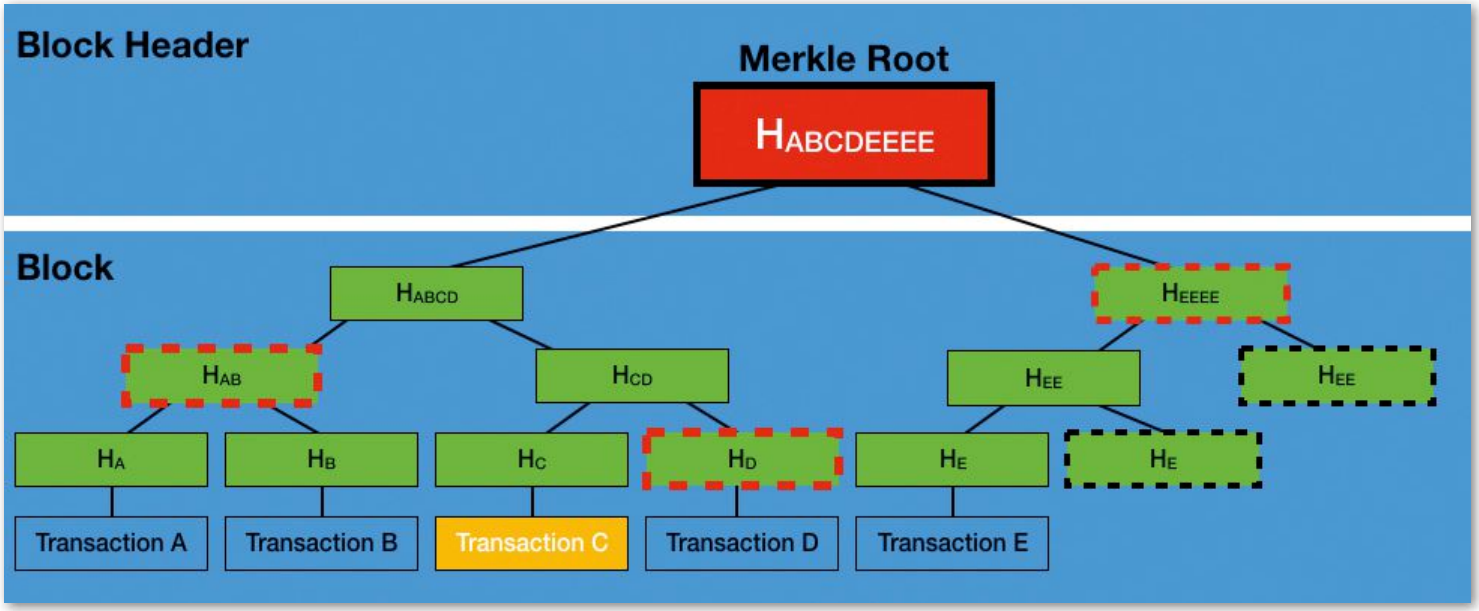


Figure 12: How the Merkle Tree and Merkle Root is used to validate a transaction

Listing 1: A Smart Contract to copyright a song's lyrics

```
pragma solidity ^0.4.17;

contract SongsCopyright {
  //---mapping is an associative array, similar
  // to key/value pairs
  // key is bytes32 and value is boolean---
  mapping (bytes32 => bool) private proofs;

  //---stores the hash of the lyrics in the
  // mapping---
  function storeProof(bytes32 proof) private {
    proofs[proof] = true;
  }

  //---calculate and store the hash (proof) for
  // a song's lyrics---
  function copyrightLyrics(string lyrics) public {
    bytes32 proof = lyricsHash(lyrics);
    storeProof(proof);
  }

  //---helper function to get a lyrics's sha256---
  function lyricsHash(string lyrics) private
    pure returns (bytes32) {
    return sha256(lyrics);
  }

  //---check if a lyrics has previously been
  // saved---
  function checkLyrics(string lyrics) public
  view returns (bool) {
    bytes32 proof = lyricsHash(lyrics);
    return hasProof(proof);
  }

  //---returns true if proof is found---
  function hasProof(bytes32 proof) private view
  returns(bool) {
    return proofs[proof];
  }
}
```

The above means that the contract compiles with a compiler version beginning with 0.4.17; but it won't work with version 0.5.0 or higher.

Next, to define a contract, you use the **contract** keyword:

```
contract SongsCopyright {
}
```

This is very much like declaring a class in languages like C# or JavaScript. Next, you declare a **state variable** named **proofs**:

```
mapping (bytes32 => bool) private proofs;
```

Think of state variables like class members in a typical programming language. In this example, the state variable is of type **mapping**, which is really like an associative array (commonly known as a dictionary). In this example, it contains key/value pairs, of which the key is of type **bytes32** (32 bytes of raw data) and its associated value of type **bool** (Boolean). For this example, you'll hash the lyrics of a song using SHA256 (which returns a 32-byte hash) and then store the hash in **proofs**. State variables values are permanently stored in contract storage.

Next, define a function called **storeProof()**, which takes in a single argument of type **bytes32**. It's declared with the **private** visibility modifier to indicate that it will be used internally within the contract and not visible outside this contract.

```
//---stores the hash of the lyrics in the
// mapping---
function storeProof(bytes32 proof) private {
  proofs[proof] = true;
}
```

The next function is called **copyrightLyrics()** and it calls the **lyricsHash()** function to hash the given song's lyrics. Once the hash is derived, it calls the **storeProof()** function to store the hash in the **proofs** mapping. Note that this function is declared with the **public** keyword to indicate that this function is callable outside the contract.

> Because State variables store values permanently on the blockchain, it's expensive to use and consumes gas whenever you need to change their values. This is because of the way the state variables store their values in the blockchain. (An explanation of this is beyond the scope of this article.)

```
//---calculate and store the hash (proof) for
  // a song's lyrics---
  function copyrightLyrics(string lyrics)
  public {
    bytes32 proof = lyricsHash(lyrics);
    storeProof(proof);
  }
```

The **lyricsHash()** function performs a hash on the song's lyrics using the **sha256()** function. Note that it's declared with the **pure** keyword. The **pure** keyword indicates that this function won't access nor change the value of state variables.

```
//---helper function to get a lyrics's
// sha256---
function lyricsHash(string lyrics) private
  pure returns (bytes32) {
  return sha256(lyrics);
}
```

The next function is **checkLyrics()**, which is declared with the **view** keyword. The **view** keyword indicates that this function accesses the value of state variables, but it never modifies them. This function takes in a song's lyrics and hashes it. It then calls the **hasProof()** function to see if the hash exists in the **proofs** mapping.

```
//---check if a lyrics has previously been
// saved---
function checkLyrics(string lyrics) public
view returns (bool) {
  bytes32 proof = lyricsHash(lyrics);
  return hasProof(proof);
}

//---returns true if proof is found---
function hasProof(bytes32 proof) private view
returns(bool) {
  return proofs[proof];
}
```

Compiling the Smart Contract

To compile a Smart Contract, you can use the **solc** compiler. The easiest is the online Solidity compiler (**Remix IDE**) available at https://remix.ethereum.org. When you first load the Remix IDE, you'll see a default contract called **ballot.sol**. Simply overwrite it with the **SongsCopyright** contract, as shown in **Figure 13**.

What is MetaMask?

MetaMask is a bridge that allows you to connect to the Ethereum network. It allows you to run a blockchain app in your browser without running a full Ethereum node. MetaMask includes a secure identity vault, providing a user interface to manage your identities on different sites and sign blockchain transactions.

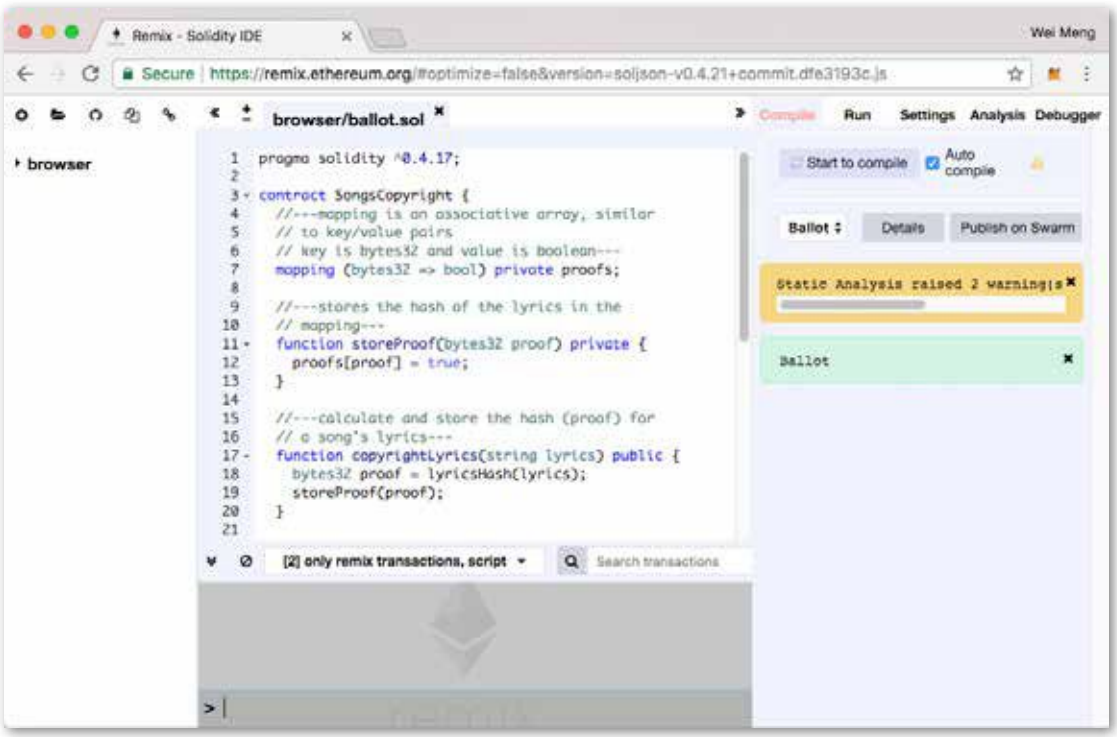


Figure 13: Use the Remix IDE to compile a smart contract.

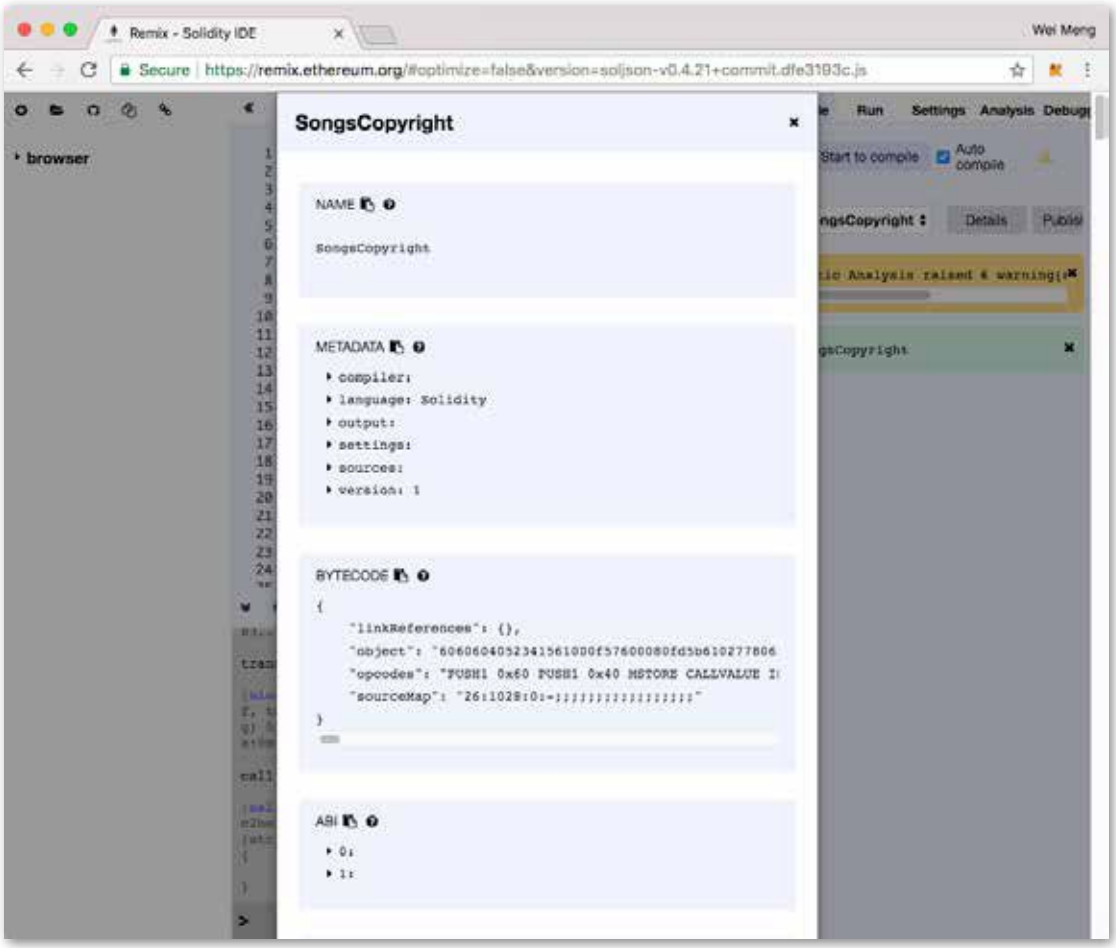


Figure 14: Getting the details of the compiled smart contract

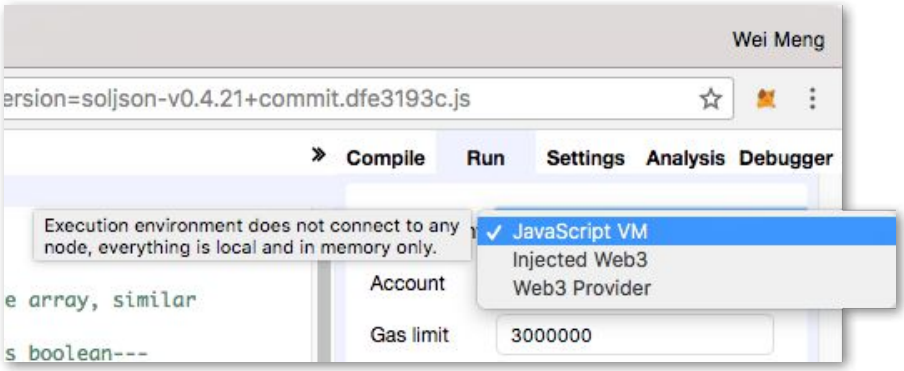


Figure 15: The Remix IDE supports three different modes of testing your smart contract.

On the right side of the window, you'll see a tab named **Compile**. Be sure to check the **Auto compile** option. This allows your contract to be compiled as you type and you can fix any errors on the fly. Quite often, there may be warning messages, but most of the time the warnings are more informational than critical. On the same tab, you'll find the **Details** button. If you click the Details button, you'll see a pop-up, as shown in **Figure 14**.

For a contract to be callable by client applications, a client application needs the following two pieces of information:

- The **ABI** (Application Binary Interface) of the contract
- The address of the deployed contract

For the first one, you can find the ABI of your contract by clicking on the icon displayed next to the ABI section, as shown in **Figure 14**. This copies the ABI of the contract into the clipboard. Paste the ABI into a text editor because you'll need it later in this article when you build a client application to interact with the contract.

For the address of the contract, you'll get it once it has been mined and added to the blockchain. You'll see this later in this article.

Testing the Contract Using the JavaScript VM

Once the smart contract is compiled without any errors, it's time to test it. The Remix IDE offers three modes of testing your smart contract:

- **JavaScript VM** simulates running your smart contract without actually deploying it onto the blockchain.
- **Injected Web3** uses a plug-in such as MetaMask in your Web browser to inject a **web3** object (see the next section for more information) so that your smart contract can be associated with an account.
- **Web3 Provider** connects directly to an Ethereum

node so that your smart contract can be associated with an account. Requires you to run an Ethereum node such as **geth**.

For this section, let's select the **JavaScript VM** (see **Figure 15**) located under the **Run** tab.

You can now click the **Create** button to simulate deploying the contract onto the blockchain. Immediately, you see the contract with two buttons (see **Figure 16**). Observe that the **checkLyrics** button is blue and the **copyrightLyrics** button is red. Blue-colored buttons represent functions that do not consume gas when called, and those in red require gas. Because the contract is simulated in this example, you won't see the difference between the two buttons, for now.

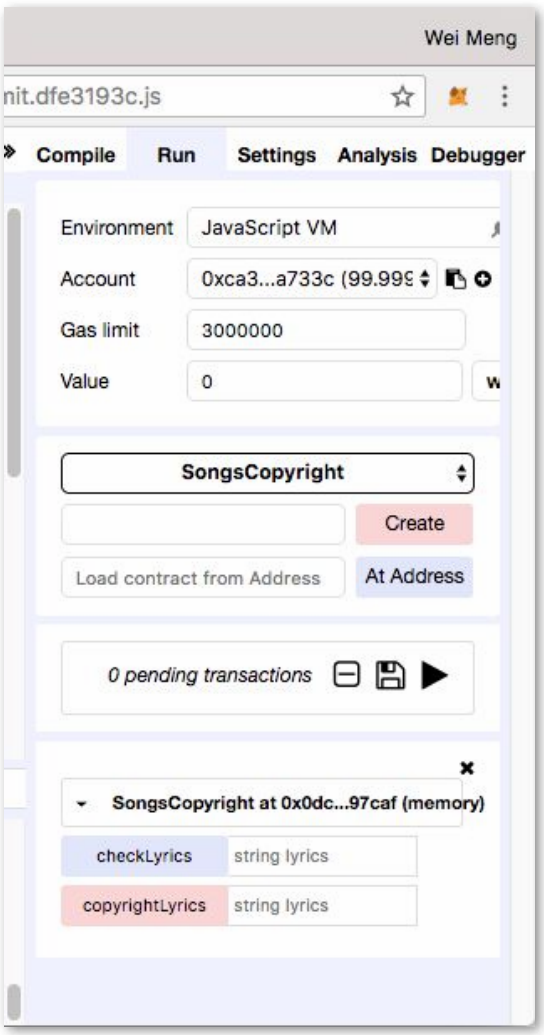


Figure 16: Test the smart contract using the JavaScript VM.

> Gas is the internal pricing for running a transaction or contract in Ethereum.

Let's now try to copyright the lyrics of a song and type in the following lyrics (together with the quotation marks) into the box displayed next to the **copyrightLyrics** button (see **Figure 17**):

"We tried to get along, we tried to just get through but something here is wrong, don't tell me this is true. There's no reason why, I never saw the sign, you didn't say goodbye. I hoped you were mine, waited up all night long, waited up all night long, waited up all night long, waited up all night long, the night went on and on. The sun was rising slow somewhere in the dawn, the saddest feelings grow, the power of that pretty face, my heart could end for you causing such an empty space. Don't tell me this is true, waited up all night long, waited up all night long, waited up all night long, waited up all night long, waited up all night long."

Click the **copyrightLyrics** button and the lyrics are now passed to the contract and the hash saved.

To verify whether the same lyrics have previously been saved into the contract, type in the same lyrics and click the **checkLyrics** button (see **Figure 18**). You should see the result as **true**.

If you now enter some other song's lyrics, you will get a **false** for the result.

Testing the Contract Using the Injected Web3

Now that your contract is tested to run without problems, let's deploy it onto a real blockchain. To do so, you need a way to get connected to the Ethereum blockchain. You do this via an Ethereum node. The easiest way to get connected to the Ethereum network is to use MetaMask (https://metamask.io).

Behind the scenes, MetaMask connects to some Ethereum nodes hosted at infura.io. Your accounts are saved in MetaMask itself, but all your transactions are relayed

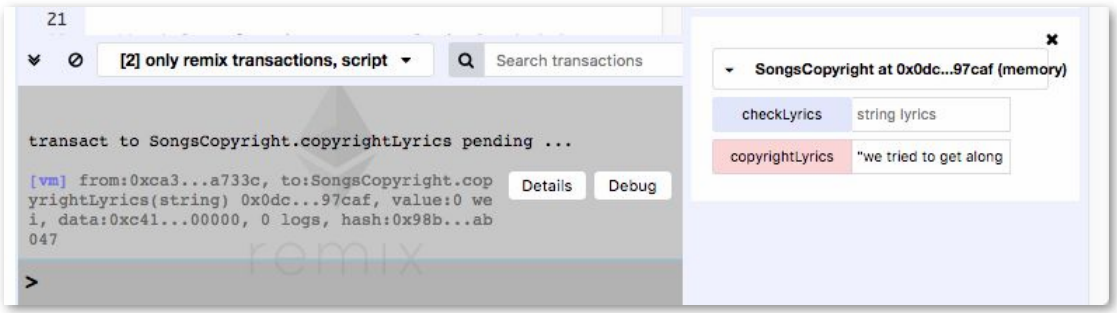


Figure 17: Send a song's lyrics to be stored on the blockchain.

What is Gas?

When a contract is deployed or is executed as a result of being triggered by a message or transaction, every instruction is executed on every node of the blockchain network. This has a cost; for every executed operation, there's a specified cost, expressed in a number of ***gas units***. Gas is the name for the execution fee that senders of transactions need to pay for every operation made on an Ethereum blockchain.

Figure 18: Verify a song's lyrics against the hash stored on the blockchain.

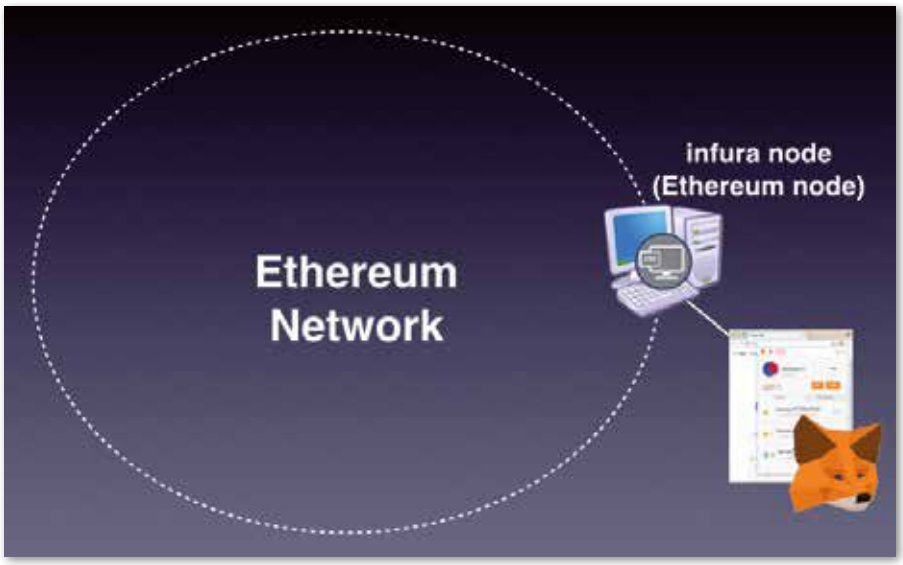


Figure 19: How MetaMask works

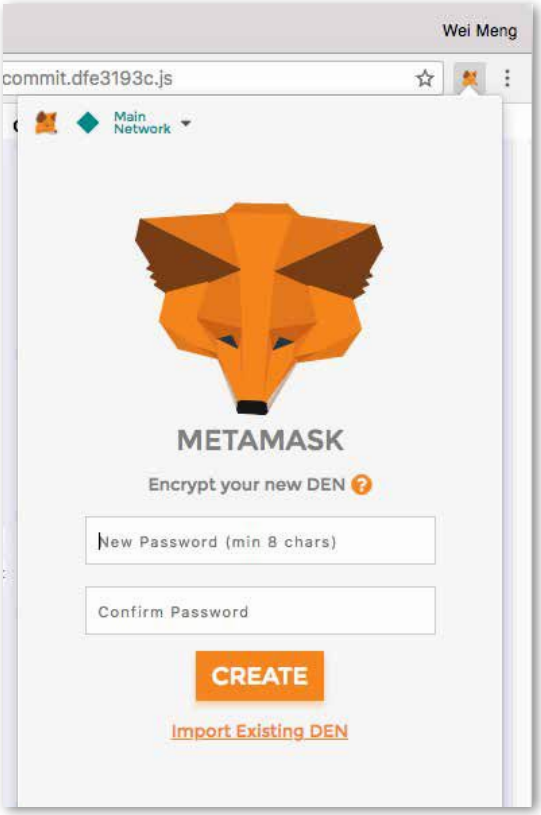


Figure 20: Enter a password so that MetaMask can secure your accounts

through the Ethereum nodes at infura.io. **Figure 19** shows how MetaMask connects to the Ethereum network.

MetaMask exists as a Chrome extension. Just launch the Chrome Web browser and navigate to https://metamask.io. Once the installation is done, you can see the MetaMask extension installed on your Chrome browser (see **Figure 20**).

Enter a password to secure your account. Once the password is entered, you will see a 12-word phrase. These 12 words allow you to restore your account(s) in the event that you have forgotten your password, or that you need to restore the accounts on another computer. Once this is done, you should see a default account created for you (see **Figure 21**). Observe your account address (in this example it's the one that says "0xc5274...").

You can also click on the drop-down item labeled **Main Network** (see **Figure 22**) to see the different Ethereum networks you can connect to. By default, MetaMask connects to the Main Ethereum Network. For development use, you should connect to one of the test networks available. Doing so spares you the need to use real Ethers for testing your smart contract. For this example, let's connect to the **Ropsten Test Network**.

> The Ethers that you obtained from the test faucet have no real monetary value, so don't be too excited if you see a monetary value assigned in MetaMask.

In order to test smart contracts on the test networks, you need Ethers. At this moment, you have none, so you need to get some. Click on the **BUY** button and then another screen will appear. Click the **ROPSTEN TEST FAUCET** button and you will be redirected to the https://faucet.metamask.io/ page (see **Figure 23**). Click the **request 1 ether from faucet** button a few times to request for some free Ethers.

After a while (an Ethereum block typically takes about 14 seconds to be mined), you should see some Ethers (see **Figure 24**).

You are now ready to deploy the contract to the test network. Back in the **Remix IDE**, select the **Injected Web3** environment. Make sure that the **Account** now displays the account you observed in MetaMask (see **Figure 25**). If you don't see the account, refreshing the page will usually fix the problem.

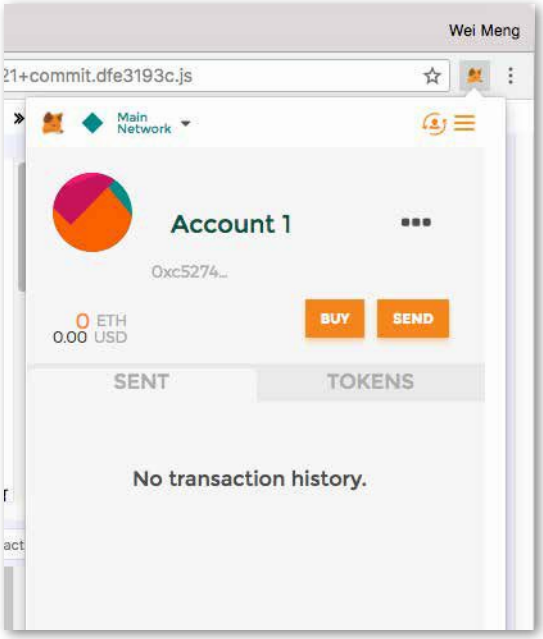


Figure 21: Your first account in MetaMask

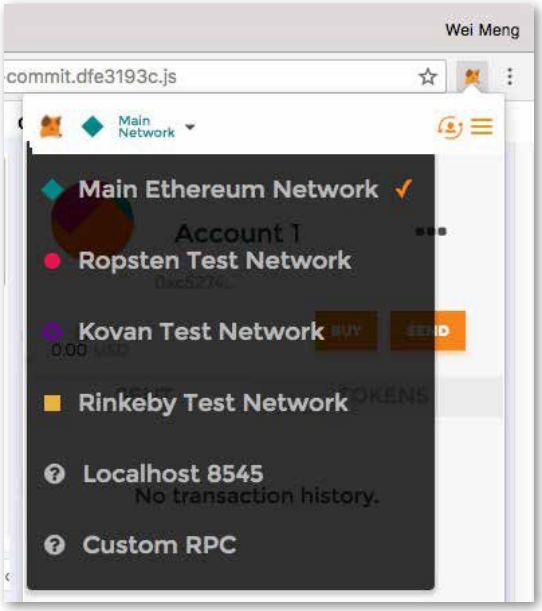


Figure 22: MetaMask can connect to the different Ethereum networks.

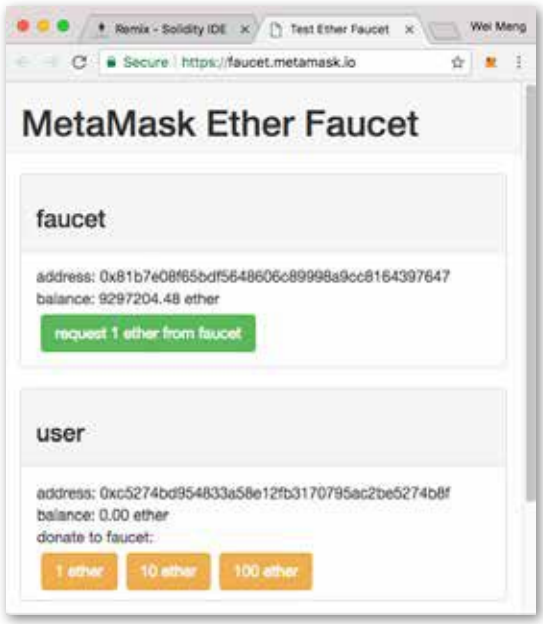


Figure 23: Request free Ethers to use on the test networks.

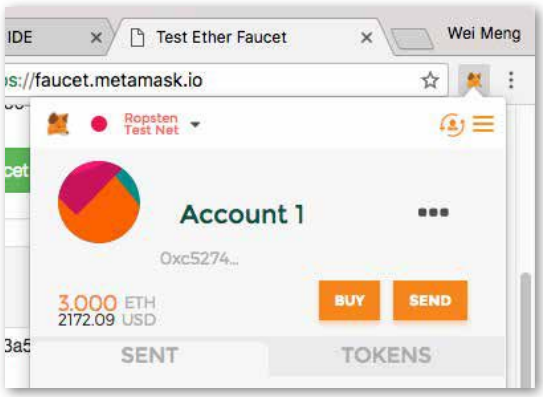


Figure 24: The account credited with some Ethers

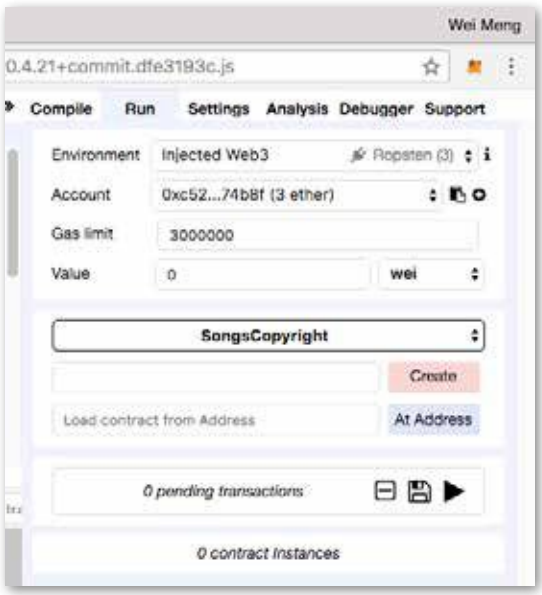


Figure 25: Test the smart contract using the Injected Web3 method.

Click the **Create** button to deploy the contract. This time around, notice that MetaMask pops up a window asking you to confirm the transaction (see **Figure 26**), with the **Gas** information specified.

Gas and Ether are decoupled deliberately; units of gas are aligned with computation units, and the price of Ether fluctuates as a result of market forces. The price of gas is decided by the miners, who can refuse to process a transaction with a lower gas price than their minimum limit. To get gas, you simply need to have Ether in your account. Ethereum clients automatically use your Ether to purchase Gas. Ether is deducted from the Ethereum account sending the transaction. The amount of gas needed for a transaction is determined by the complexity of

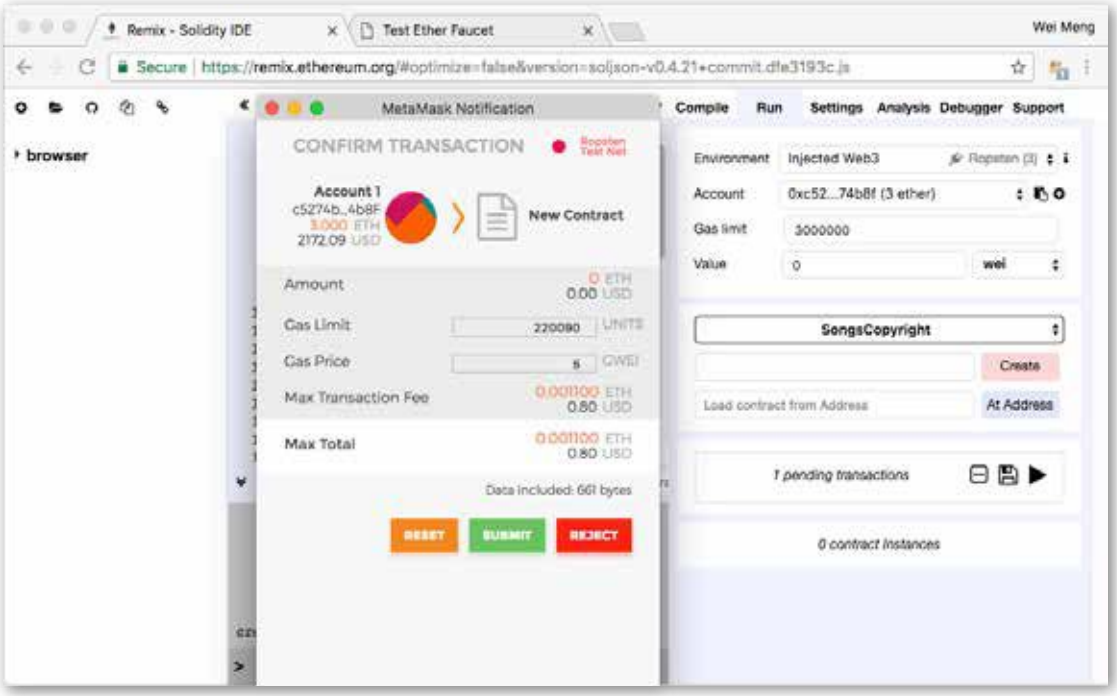


Figure 26: Deploying a smart contract consumes gas.

Figure 27: The smart contract is now deployed on the Ropsten test network.

the contract. You specify the gas limit (min. 21,000 gas) to indicate the maximum amount of gas you are willing to spend on a contract. This prevents you from spending an unlimited amount of gas on a contract that runs indefinitely (due to a bug). All unused gas is refunded back to you. As shown in **Figure 26**, a maximum of 220090 gas is needed and a unit of gas is priced at 5 GWEI (1 Ether is equal to 1000000000 GWEI), giving it a total of (220090 * 5)/1000000000 = 0.0011 Ether. Based on the time of this writing, this is worth about $0.80.

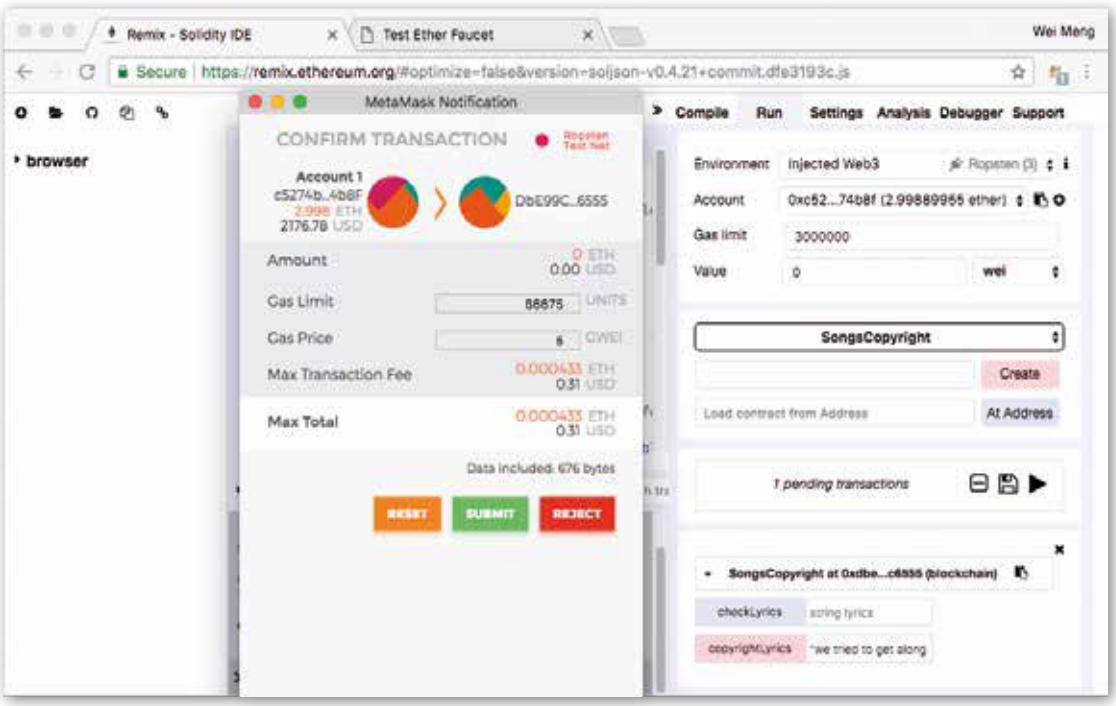


Figure 28: Sending a song's lyrics to the deployed smart contract requires gas.

Click the **SUBMIT** button to confirm the transaction. When the transaction is mined, you'll see the contract deployed together with the contract address (see **Figure 27**).

You can now enter the song lyrics and then click the **copyrightLyrics** button. I mentioned earlier that buttons that are red require payments. In this case, clicking on the **copyrightLyrics** button brings up the confirmation window again, as shown in **Figure 28**.

Click the **SUBMIT** button and the transaction will be added to a block and mined. After a while, you'll be able to enter the same lyrics in the textbox next to the **checkLyrics** button to verify the lyrics. Observe that for this transaction, no gas is required, as the request isn't modifying the state variables in the smart contract. Once the transaction is mined and the block containing it is added to the blockchain, the hash of your song's lyrics is forever available in the blockchain and remains immutable.

Writing a Web Application to Invoke the Smart Contract

In the previous sections, you tested the smart contract and deployed it onto the Ropsten test network. You even had the chance to test it in the Remix IDE. However, in real life usage, it isn't practical to ask your user to use Remix IDE. A better way should exist to interact with your smart contract, ideally through a Web application.

To interact with a smart contract from within a Web application, you can use the **web3.js** APIs. The web3.js is a collection of libraries that allow you to interact with a local or remote Ethereum node, using a HTTP or IPC connection.

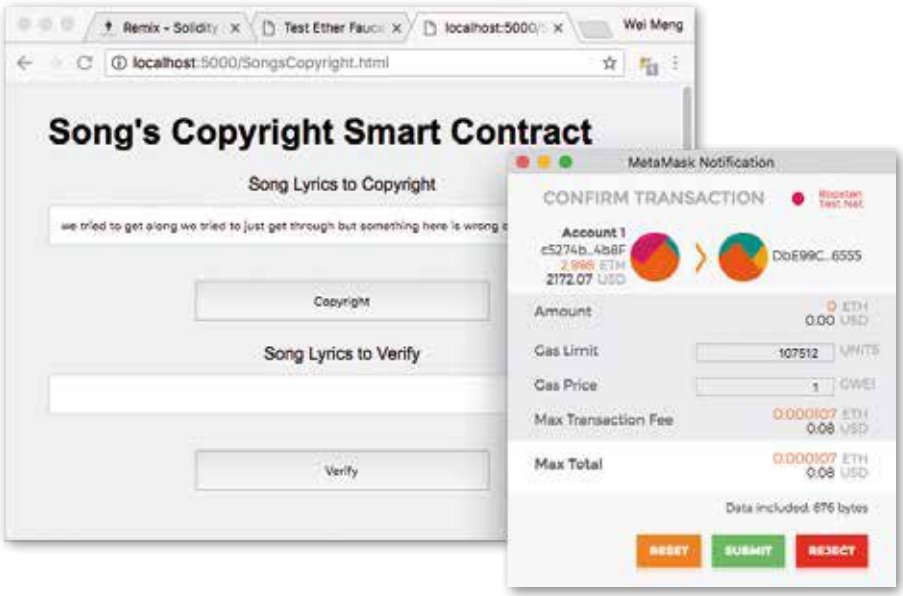


Figure 29: The Web application is automatically linked to the account in MetaMask.

Listing 2: The content of SongsCopyright.html

```
<html lang="en">
    <head>
        <link rel="stylesheet" type="text/css"
              href="main.css">
        <script src="./node_modules/web3/dist/web3.min.js">
        </script>
    </head>
    <body>
    <div class="container">
        <h1>Song's Copyright Smart Contract</h1>
        <center>
            <label> Song Lyrics to Copyright</label>
            <input id="songlyricstocopyright" type="text">
            <button id="btnCopyright">Copyright</button>

            <label>Song Lyrics to Verify</label>
            <input id="songlyricstoverify" type="text">
            <button id="btnVerify">Verify</button>
        </center>
    </div>
    <script
        src="https://code.jquery.com/jquery-
             3.2.1.slim.min.js"></script>
    <script>
        if (typeof web3 !== 'undefined') {
            // when using Metamask, web3 would be injected
            web3 = new Web3(web3.currentProvider);
        } else {
            // set the provider you want from Web3.providers
            // this block will be executed if you are not
            // using Metamask
            web3 = new Web3(new
                Web3.providers.HttpProvider(
                "http://localhost:8545"));
        }

        //---the ABI of the contract---
        var contractABI = web3.eth.contract(
            [ {   "constant": true,   "inputs":
            [    {     "name": "lyrics",
            "type": "string"    }   ],   "name":
            "checkLyrics",
            "outputs": [    {     "name": "",
            "type": "bool"
            }   ],   "payable": false,   "stateMutability":
            "view",   "type": "function"  },  {
            "constant": false,   "inputs": [
            {     "name": "lyrics",     "type": "string"
            }   ],   "name": "copyrightLyrics",
            "outputs": [],
            "payable": false,   "stateMutability":
            "nonpayable",
            "type": "function"  } ]);

        //---replace the contract address with your own---
        var contract = contractABI.at(
            '0xdbe99ce4ffd917796d85e134814c5fae625c6555');

        $("#btnCopyright").click(function() {
            var songlyrics = $("#songlyricstocopyright").val();
            console.log(songlyrics);
            contract.copyrightLyrics(
               songlyrics,(err, result) => {});
        });

        $("#btnVerify").click(function() {
            var songlyrics = $("#songlyricstoverify").val();
            contract.checkLyrics(songlyrics, (err, result) => {
                $("#result").html(result);
                alert(result)
            });
        });
        </script>
    </body>
</html>
```

Listing 3: The content of main.css

```
body {
    background-color:#F0F0F0;
    padding: 2em;
    font-family: 'Arial';
}
label {
    display:block;
    margin-bottom:10px;
}
input {
    padding:10px;
    width:100%;
    margin-bottom: 1em;
}
button {
    margin: 2em 0;
    padding: 1em 4em;
    width: 50%;
    display:block;
}
```

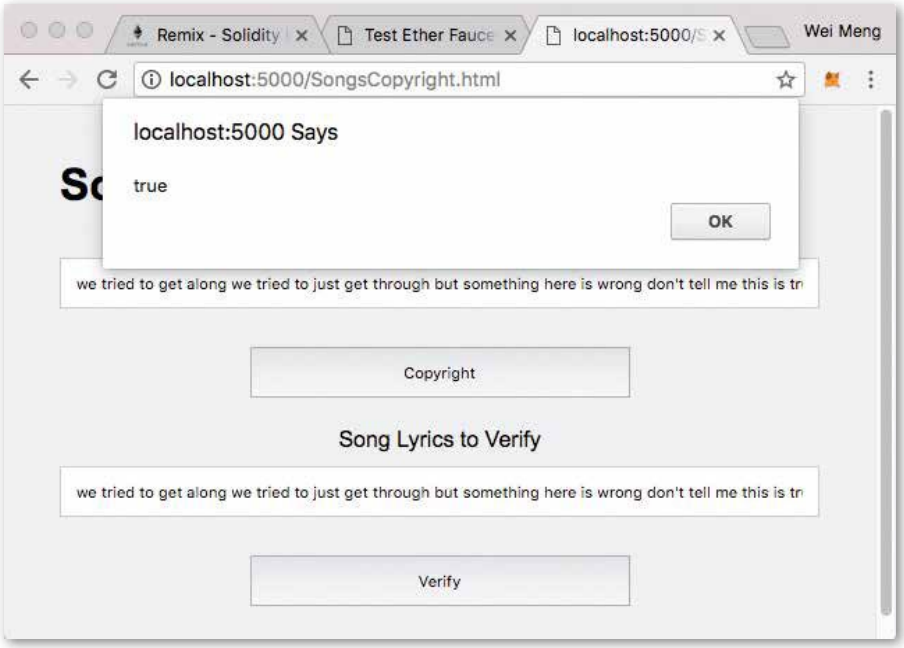


Figure 30: Verifying a song's hash on the smart contract

To see how it's used, let's now create a Web application to interact with the Smart Contract that you've just deployed to the Ethereum Ropsten test network.

Type the following commands in the Terminal (you need to install Node.js in order for the **npm** command to work):

```
$ cd ~
$ mkdir UseContract
$ cd UseContract/
$ npm init
[Press Enter a few times]
```

The above commands create a directory named **UseContract** in the home directory of your account. You then use **npm** to initialize a new project. Once that's done, type the following command:

```
$ npm install --save ethereum/web3.js
```

This downloads the web3.js module and its dependencies. Remember when you used MetaMask in the earlier section of this article? MetaMask injects the web3.js JavaScript library to allow you to carry out transactions via a regular Web page in Chrome. This means that when your Web application interacts with a Smart Contract, it automatically links to your account in MetaMask so that it can use the account to pay for gas, as well as send Ether to another user or account.

In order for web3.js to work correctly (due to security concerns), your HTML pages must be served from **http://**, and not **file:///**. If you have a Web server, you can store the Web pages in the Web publishing directory of your computer. For development purposes, there is a Node.js module that allows you to serve a HTML page from wherever it's saved. Type the following command to install the **serve** module:

```
$ npm install -g serve
```

Let's now create the HTML page to interact with the Smart Contract. Create a file named **SongsCopyright.html** and save it in the **UseContract** folder. Populate it as shown in **Listing 2**. Note that to invoke a contract, a client needs the ABI of the contract as well as its address.

Create another file named **main.css** and save it in the **UseContract** folder. Populate it as shown in **Listing 3**.

To start the Web server, type the following commands in Terminal:

```
$ cd ~/UseContract
$ serve
```

Using the Chrome browser (with the MetaMask extension installed), load the HTML page using http://localhost:5000/SongsCopyright.html/. You can now enter the song lyrics and then click on the **Copyright** button. The MetaMask window now automatically pops up asking you to submit or reject the transaction (see **Figure 29**).

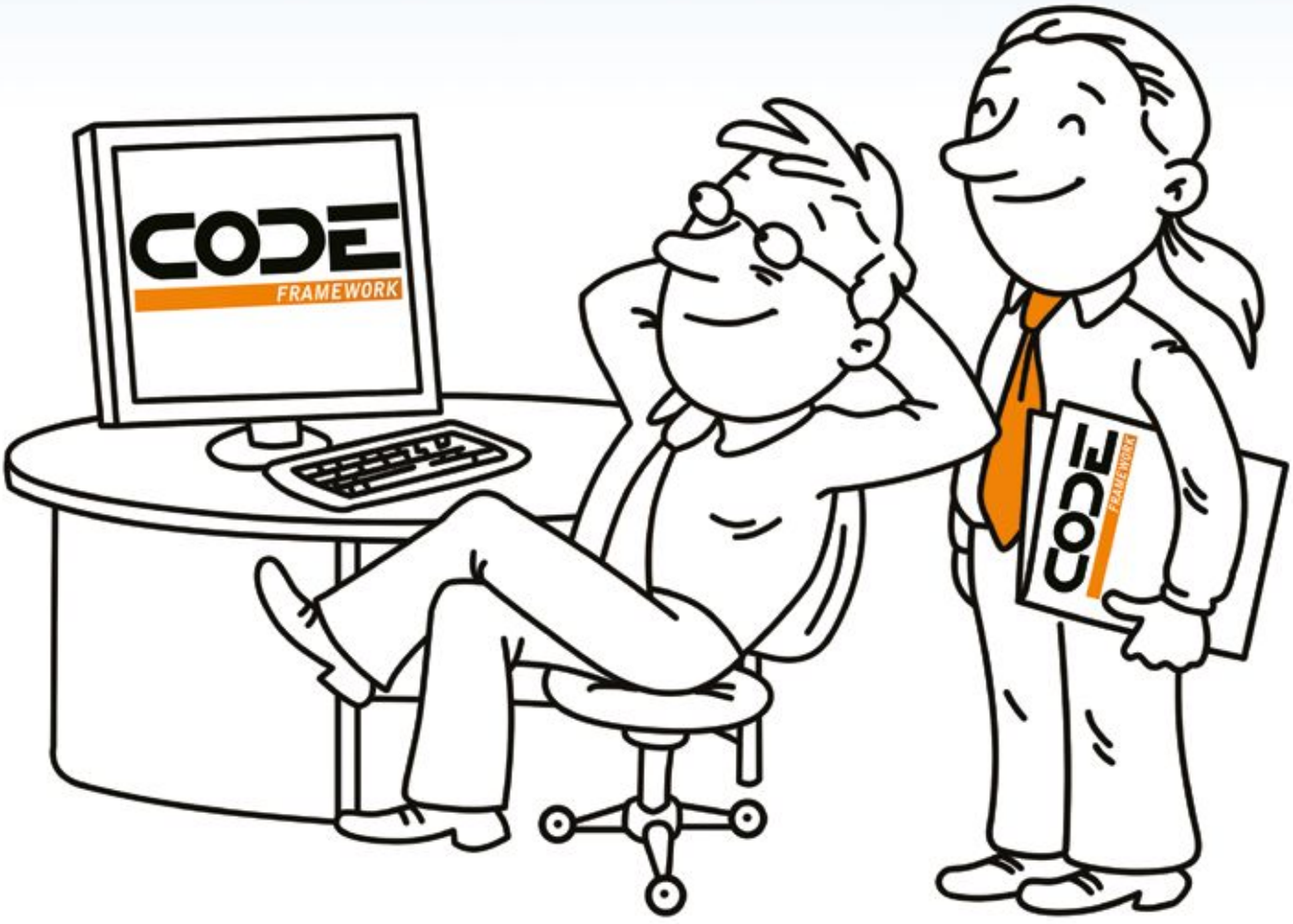
Once the transaction has been confirmed and the block mined, you can now verify the song lyrics. Clicking on the **Verify** button displays the result as an alert (see **Figure 30**).

Summary

I hope that I've provided an easy way for you to understand Blockchain and see how it works. Although there are many details that I haven't discussed in this article due to the constraint of space, it should help you get started with Blockchain, in particular with Smart Contracts, and to see how it can be used for a lot of real-world applications. We are still in the early days of this technology, so prepare to see many more creative uses of Blockchain and smart contracts coming your way in the near future!

Wei-Meng Lee

CODE Framework:
Business Applications Made Easy



Architected by Markus Egger and the experts at CODE Magazine, CODE Framework is the world's most productive, maintainable, and reusable business application development framework for today's developers. CODE Framework supports existing application interoperability or can be used as a foundation for ground-up development. Best of all, it's free, open source, and professionally supported.

Download CODE Framework at www.codemag.com/framework

Helping Companies Build Better Software Since 1993

www.codemag.com/framework
832-717-4445 ext. 9 • info@codemag.com

CODE
FRAMEWORK

Docker

It's not often that an open-source tool or technology emerges and comes to a position of industry mainstream acceptance within a span of two years. Docker is one such technology that fits within that rarified category, and it deserves discovery and discussion. However, owing to Docker's roots, it often remains entirely a mystery to those developers who find themselves among "the Microsoft crowd," particularly when the root of Docker's benefit—that of providing isolation at a computer- or operating-system-level—already seems to be covered by Hyper-V and its ilk. Why all the hubbub? What compelling capability does Docker offer that managing Hyper-V virtual machines doesn't?

Ted Neward

Ted Neward is the CTO of iTrellis, a software development consultancy located in Seattle. He's an internationally recognized speaker, instructor, consultant, and mentor and spends a great deal of time these days focusing on languages and execution engines like the JVM and CLR.

This article seeks to do two things: one, to get the Docker neophyte up and running with it, through most of the "basics" that you'd need to understand in order to use it for several simple purposes; and two, to understand how Docker differs from what appear to be its contemporary competitors/alternatives, and why Docker serves as a useful tool beyond the traditional Hyper-V story.

Let's begin.

Installing

Getting Docker onto your system is pretty trivial; by visiting the Docker home page, you can download the official Docker installer for Windows. The Docker download is available at https://docs.docker.com/install/. This installs the official "Docker for Windows" release, which will, among other things, start a process running in the background on your computer. This is the Docker daemon, and it's the daemon that does the majority of the work. When you use the Docker command-line tool, it sends instructions to the daemon to carry out. If, for any reason, the daemon isn't running, the Docker command-line tool errors out without being able to do any meaningful work.

One thing that's important to note about Docker for Windows: Unlike Docker on its original platform (Linux), Docker for Windows requires the use of some virtualization technology to carry out its agenda. If you have Hyper-V enabled, Docker for Windows can use that; however, older versions of Docker used VirtualBox (an open-source virtualization tool now managed and maintained by Oracle), and that is also usable, if you desire. The key thing to understand is that Hyper-V and VirtualBox cannot both be enabled on the same computer at the same time—it's a choice of one or the other, never both. (This is due to limitations in the virtualization stack, and not something that's easily fixable at this time.)

Assuming that the installation has worked, Docker for Windows is now running on your computer. To verify that it's installed correctly and running, the easiest thing to do is ask it to report the version installed:

```
docker -v
```

Docker responds with something similar to the following:

```
Docker version 17.09.1-ce, build 19e2cf6
```

Docker's version numbering scheme informs you of major and minor versions, and the fact that this is the "Community Edition" (hence the **ce** suffix). It also reports, as you can see, the build number, but this is generally of little use except in error log reports in the event something goes wrong.

Exploring

To begin understanding the promise and potential of Docker, let's use it to do something.

In particular, let's do the traditional "Hello World" of Docker, which is to run a simple **bash** command in an Ubuntu Docker image:

```
$ docker run hello-world
Hello from Docker!
This message shows that your installation appears to
be working correctly.

To generate this message, Docker took the following
steps:
( ... snip ... )
```

Docker's hello-world image is cute, but hardly functional. Let's do something slightly more ambitious. The Docker hello-world image suggests running a bash shell inside of an Ubuntu image, but you can do something a little different and closer to home:

```
docker run -it microsoft/dotnet
( ... snip ... )
root@83ea2918a67f:/#
```

What's happening here is both obvious and deceptive: The Docker daemon downloads a collection of filesystem images to your computer. Each filesystem image is, at heart, a delta of a previous image; in essence, a Docker image is made up of a series of filesystem diffs, in the same way that Git and other source-control systems have been keeping track of the changes to your source code for years. Each filesystem delta is downloaded, applied in order, and the end result...well, the end result is an executable environment in which you can run a command. In this particular case, the command is implicitly **bash**, which is the standard command shell in this particular image, and, because this is the standard Microsoft .NET Core Docker image, it has all of the .NET Core tools installed in it. Specifically, you can "dotnet help" or any of the other .NET Core commands, including the generation of a new .NET project, should you choose. There's not much point to doing that at the moment, as there are a number of other things you need to do first, but the point is this: Docker can act as a virtualization tool, providing an environment in which you can execute commands and run processes. Or, to be more precise, a single process. But more on that in a moment.

The command-line you used is important to parse. Like so many other command-line tools, a Docker command consists of a verb (**run**) and a target, in this case, the image to run (**microsoft/dotnet**). That image is stored in a cloud Docker repository called the Docker Hub, and it's accessible to anyone who wishes to use it. This is the central repository for all public Docker images, and holds the "official" images for a number of different tools, not just .NET Core. Numerous open-source tools, such as OpenJDK, MongoDB, Neo4J, Postgres, and others, are stored here, always available for use.

> Docker can act as a virtualization tool, providing an environment in which you can execute commands and run processes.

The naming convention takes one of two forms. If the image is an "official" DockerHub image, it has only one part to the name, such as **mongodb** or **postgres**. If the image is managed by a third party, like, in this case, Microsoft, it forms a two-part name, such as **microsoft/dotnet**. The third-party status isn't reserved exclusively for big companies. As a matter of fact, should you feel compelled, you can run the exact same command using one of the Docker images that I've hosted on DockerHub, such as my image that hosts Common Lisp.

```
docker run -it tedneward/clisp
root@83ea2918a67f:/#
```

Ditto for any other of my images: **rust**, **ooc**, and **purescript**, plus a few more mainstream ones. The point is, the images are stored in DockerHub, ready for anyone to use.

Assuming that you decide you'd like to spend some time with Lisp, you could make use of my Common-Lisp Docker image. You can run the above, but there's a huge problem with that; any changes you make to the filesystem are wiped out as soon as you exit the Docker image. Surely there has to be a way to preserve those changes.

In the Docker world, this means "mounting a local volume." Put differently, you want to tell Docker to mount a directory on the computer to a directory point inside the Docker image. Contrary to expectations, the mount command isn't a part of the Docker image, but something that you set when you start the image itself. In other words, you need to pass another command-line parameter to tell the Docker image where you want to mount the volume on the host (the physical computer) and where you want its mount point to be in the image.

In this particular case, I've created the image to have an empty **home** directory (**/home/clisp** inside the image) so that I have a place to mount a directory from the host computer. Doing that requires the use of the **volume** parameter in Docker, which looks something like this (because of the limitations of the print column, the line is broken—when you're using it, be sure that it's all on one line):

```
docker run -it -v $(pwd):/home/clisp tedneward/clisp
clisp@50c812137d8b:~$
```

Now, any changes that you make to your current-working-directory are reflected inside the Docker image, and vice versa. For example, if I fire up Visual Studio Code as an editor from the host and create the following file:

```
(defun hello (name)
   (format t "Hello, ~a" name))

(hello "Ted")
```

Then save it to the current directory as **hello.cl**. Inside of the Docker image (which you previously fired up using the **run** command), you can run the C-Lisp code and see the lovely results:

```
clisp@98a2e68ea7a1:~$ clisp hello.cl
Hello, Ted
clisp@98a2e68ea7a1:~$
```

Congratulations! You're now officially a Lisp programmer and can add that to your resume. What's more, the image is stored to your local hard drive, so you can fire it up any time you like, including when you're offline. When you exit the image, any files stored in the **/home/clisp** directory (or any of its subdirectories that you might create along the way) in the image will be written to the host filesystem and thus preserved after the image terminates. The Docker documentation has a great deal more to say on this subject, including some suggestions for how best to manage databases that store a ton of data, but this suffices to make the point that Docker filesystems are ephemeral, and only mounted volumes persist beyond the image's execution. On the surface of it, that seems odd for a virtualization technology.

What's also odd is that you can't do anything inside the image that isn't restricted to a terminal shell. You can't fire up VSCode inside the image, for example, because there's no GUI system inside the image by which to display the VSCode windows. This, too, seems odd for a virtualization tool. It's almost as if we're back to the 1990s, with terminal shell access over the network as the only way to reach a computer. Even a decade ago, you could use remote-connect tools like VNC or RDP to connect GUIs to remote computer, so what gives? (Hold onto that question for a moment.)

One Docker for Windows quirk also deserves mention. Docker for Windows can only use host drives/directories that are available under Windows Filesystem Sharing. By default, the Docker for Windows installation process tries to automate this so it's seamless to the developer, but if you try to use a host directory on a different drive than the boot (C:) drive, you can start to run into errors from the client. If you run into issues, try mounting a host directory under your user account's home directory (C:/Users/Ted/...) to see if that corrects the problem.

Even without these limitations, Docker wouldn't be all that useful if the only thing it could do is mount local filesystems; fortunately, Docker can also open ports in the image that can connect to actual ports on the host.

Doing so requires another mapping parameter, **p**, which maps the port on the image to the port on the host. Common-Lisp is wonderful, but most readers are going to be more familiar with .NET Core, so let's turn back to the official Microsoft .NET Core image, create a new ASP.NET MVC Web app with only the scaffolded files, and run it.

To do this is actually pretty straightforward. Use the Docker command-line shown earlier to launch the Docker image, but with a **volume** parameter to map to the current working-directory, and the **port** parameter to map .NET Core's default port of 5000 in the image to the host computer's port of 5050 (or any other open port of your choice). The first step is to launch the .NET Core image. (All of the following code should be typed on one line, allowing it to wrap as needed.)

```
docker run -it -v $(pwd):/home/dotnet -p 5050:5000
microsoft/dotnet
```

This brings up the .NET Core image. From here, you can **dotnet new mvc -o hello**, which scaffolds out the basic template, and then **dotnet run**, which starts listening on the image's port 5000 for incoming traffic.

Note that as of this writing, Docker for Windows and Docker for Mac still have some quirks with networking that could reasonably be called bugs. In particular, Docker doesn't seem to always do machine-IP forwarding correctly, so instead of being able to browse http://localhost:5000 to see your scaffolded ASP.NET app, you may have to discover the Docker image's IP address on the network and use that directly instead. This is one of the weaker areas of Docker, and the Docker community is working to improve the experience here, but it's not unusual for Windows or Mac developers to end up having to do a little research to get things working. What's worse, any advice I might proffer here could easily be out-of-date by the time you read this. (This is, in many ways, Docker's worst story.)

If I were to stop here, it would be an interesting—if limited—virtualization story. After all, a virtualization tool that can't track the changes to the filesystem, for example, is going to run into a number of limitations pretty quickly. And no GUI? This is not exactly a compelling discussion. Let's see what else we can do.

Building

Part of Docker's appeal is that it doesn't consist only of a virtualization layer. In many respects, Docker is about capturing not only the operating environment (such as the operating system and hardware), but also the application code and the immediate context required to run it.

Consider, if you will, the average ASP.NET application that we stand up, even something relatively simple like the ubiquitous "CRUD app over Northwind database" demo that has been the staple of every .NET column author for the last two decades. If you want to see that demo in action, frequently it requires that you download the Northwind dataset sample, stand up your own instance of SQL Server (regardless of which edition), run the scripts to install the data into the database, download the code for the demo, configure the ASP.NET configuration files to point to your particular instance of SQL Server, and so on. As much as you might like to pretend that the code is what's important, many (if not most) of the issues getting the application up and running have nothing to do with code. They are often configuration issues like connection strings, passwords, and the rest. Not to mention all the time required to do that configuration.

This is where some of the Docker peculiarities start to make sense. Docker lets us capture both the code (usually after a build is done, if compilation is required) and the configuration files as part of one of those filesystem deltas, pull the filesystem down to any Docker-enabled computer connected to a Docker repository (such as DockerHub), and deploy exactly the same bits in exactly the same way, over and over and over again.

It is, in short, the ultimate delivery artifact.

Or, to be more precise, Docker allows you to build a file that can create that ultimate delivery artifact, which can then be loaded and executed. This is partly why so much of the Docker discussion comes hand-in-hand with cloud providers. Instead of having to configure the cloud virtual machines, if the cloud VMs are simply running Docker daemons, they can each pull down the complete image and start execution from that known state. (Databases and other data-storage configurations are still a little tricky in Docker, but I'll talk about ways to handle that in a second.)

The secret to all of this is the Dockerfile, a text file containing a sequence of statements that describe how to build the image. Consider, for example, the Dockerfile from my Common-Lisp image of earlier (again, note that the **&& user add** line needs to be all on a single line and is broken here to accommodate printing):

```
FROM ubuntu

MAINTAINER Ted Neward <ted@tedneward.com>

RUN apt-get update && \
    apt-get install -y clisp

RUN groupadd --gid 1000 clisp \
    && useradd --uid 1000 --gid clisp --shell /bin/bash
--create-home clisp \
    && chown -R clisp:clisp /usr/local

USER clisp
WORKDIR /home/clisp

ENTRYPOINT ["/bin/bash"]
```

For a full reference to the Dockerfile syntax, see the Docker documentation, but in a nutshell, here's what's going on.

First, this Docker image should be based on an existing image in the Docker repository called **ubuntu**. This is, not surprisingly, a standard image of the Ubuntu operating system, and I find it easy to work with for some of the images I want to create. The drawback to Ubuntu, bluntly, is that it's large and includes a number of tools and commands that won't be necessary in your typi-

cal headless production environment. For this reason, many Docker enthusiasts prefer a different flavor of Linux called Alpine Linux, which is a bare-bones OS and not much else. Were I building an image that I wanted to ship to a cloud cluster, I'd probably prefer that approach.

The **MAINTAINER** line is simply documentation that has no effect on the image.

Each of the **RUN** lines are commands executed in the newly forged image. The first asks the standard Ubuntu packaging manager to update itself and install the Common-Lisp package. The second takes the time to create a user and group on the filesystem that will be specific for clisp; it's not necessary, but by default these commands (and the process that you launch) run as the root user inside the image. Given that this image is intended for interactive (rather than production) use, I'd prefer to not be the root while I'm toying around with Lisp. More importantly, each of these commands is "checkpointed" inside the image, so there's a marginal set of benefits to not having too many of these **RUN** commands inside a single Dockerfile.

> For any public image in DockerHub, you can click on the "Dockerfile" link on its home page and see the syntax of the Dockerfile that built that image. Feel free to poke around!

The **USER** command tells Docker to switch over to using the **clisp** user instead of root, and **WORKDIR** indicates that the working directory when the image launches its process should be the **/home/clisp** directory. Again, this is appropriate for an interactive image like the one I'm building here. Lastly, the **ENTRYPOINT** directive tells Docker to run **/bin/bash**, the interactive shell, as the command to launch the image.

The key thing to understand here is that a Dockerfile consists of basically two things: a series of commands and statements about how to set up the environment, and a single command to launch a single process. This is what happened earlier when you first ran the Docker hello-world image; the image did the setup necessary to get to the point of being able to run the command that printed "Hello from Docker", and then the process terminated and Docker returned control back to you. Often, these Docker processes are long-running ("infinite") processes like a Web server that should never terminate. In fact, that's so often the case that Docker assumes you don't want to interact with the running Docker image by default, and you have to tell Docker to run interactively (the **-i** parameter to the **docker run** that you used earlier) and to connect Docker to the command prompt's stdin and stdout (the **-t** parameter).

Let's look at a more reasonable example of Docker use for an application.

Normally, you'll do development on a local computer, where you have access to all the usual goodness that you've come to expect, *a la* Visual Studio or VSCode. Assuming that you build an ASP.NET Core application, you'll ask it to publish the resulting build files to a known location on your computer, then tell Docker to build an image consisting of those build files. (The actual ASP.NET app doesn't matter for this scenario—anything will do.)

First, to generate the ASP.NET Core app, let's just do the traditional MVC app: **dotnet new mvc -o hello**, or File > New > Project from inside of Visual Studio, whichever floats your boat. (Remember, all of this is on the host computer.) Now, however, publish the app via either the Visual Studio GUI or **dotnet publish -o ./published**, which puts all the relevant files into a local "published" directory.

Next, you need a Dockerfile that tells Docker to build an image containing these bits. You could use the **microsoft/dotnet** .NET Core image, but that means you'd have to do any standard ASP.NET configuration/commands in every Dockerfile you build. Microsoft beat us to that particular punch by publishing the **microsoft/aspnetcore** image. It's probably a good idea to have your own directory inside the image that contains all of your published code, so let's create one and call it **app**, and copy all the published files there. Lastly, you'll need to tell Docker to fire up **dotnet** to run the published assembly that contains the application code, which, in the case of the **hello** app will be **hello.dll**. The Dockerfile looks like this:

```
FROM microsoft/aspnetcore:2.0

WORKDIR /app

COPY ./published .

ENTRYPOINT ["dotnet", "hello.dll"]
```

And you're done. Almost.

Docker doesn't want to have to rebuild the image every single time you tell it to run an instance of the image, so Docker wants you to compile these Dockerfiles into images. As a matter of fact, I've been a little loose with my terminology; an **image** is the filesystem contents that will be loaded into a **container** and executed. Running instances are called containers, and the thing they're running is an **image**, in much the same way that you use **objects** to refer to the running instances of things instantiated from a **class**.

You need to build this Docker image, and you do that by running **docker build .** in the directory containing the **published** directory and the Dockerfile (which, by convention, is assumed to be named "Dockerfile"). Note that if you leave out the "." from the command, Docker errors out because it needs to be told where to find the Dockerfile. It doesn't assume the current directory.

Assuming that everything works, you should see:

```
$ docker build .
Sending build context to Docker daemon  6.152MB
Step 1/4 : FROM microsoft/aspnetcore:2.0
```

```
 ---> bb8bdc966bb5
Step 2/4 : WORKDIR /app
 ---> Using cache
 ---> dd0676b321cf
Step 3/4 : COPY ./published .
 ---> c21725750682
Step 4/4 : ENTRYPOINT dotnet hello.dll
 ---> Running in 4a856a786707
 ---> 89ecf7a7917e
Removing intermediate container 4a856a786707
Successfully built 89ecf7a7917e
$
```

Yikes. It built, but Docker, by default, decided that the name of this image should be "89ecf7a7917e", which is not exactly a user-friendly name for the image. Fortunately, you can name them using the **-t** parameter to **docker build**. Thus, **docker build -t myaspnetapp .** does the same thing again, but this time calls the image **myaspnetapp**.

Now, an image called **myaspnetapp** is sitting on your local computer. But this is hardly the point—if Docker is a deployment mechanism, how do you deploy?

Publishing

The DockerHub is only one of many places to which a Docker image can be published, but most will work in much the same way, so you'll use this as an example. First, in order to make sure that your app doesn't conflict with anybody else's app on the DockerHub, the name of the image should be prefixed with your unique username on DockerHub. To do that, two things need to happen. First, you need to obtain credentials on DockerHub (go to http://hub.docker.com and create a free account), and second, the image needs to be rebuilt under the two-part naming scheme from before ("tedneward/myaspnetapp", in this case).

Once that's done, from the command-line, you can log into DockerHub by issuing "docker login" and following the prompts. Once done, the image can be uploaded to DockerHub by issuing "docker push tedneward/myaspnetapp". This does a similar set of steps to building the image, uploading each filesystem delta as necessary, and concluding by printing a hash of the uploaded image for verification. If you visit the DockerHub page for your account, the new Docker image should be uploaded, ready, and waiting for anybody else to grab and run. They just need to do as you started with: "docker run tedneward/myaspnetapp", and lo, they have a copy of the application—fully provisioned, configured, and ready to receive incoming requests—up and running. (To be fair, if it's an ASP.NET app, they'll need to configure the local-to-guest port mapping, but that's still a great deal less work than the pre-Docker alternative.)

Wrapping Up

Where does that leave us with respect to Docker?

This is where the fun begins for Docker. A Dockerfile is often be the end-step in a CI pipeline, so that the result of the build is a complete application and environment. The Dockerfile is most certainly a source artifact, so it needs to be kept in sync with any changes made to the project—if you introduce a configuration file, for example, you'll need to make sure it's reflected somewhere inside the Docker image—and needs to be checked into source control so that the CI pipeline can invoke "docker build" to produce the resulting image.

But it gets more interesting from there. If your project, like so many, is a distributed system, each node in the system can be represented by a separate and standalone Docker image. Your database, your messaging server, even a file server, these can all be generalized down into Docker images, which gives you tremendous flexibility in how the system runs during development—parts of it can be local to your computer, and other parts are in a cloud environment, and so on. This forces you to deal with how to "discover" those nodes far earlier in the project, which has the lovely side-effect of making your project more resilient in the face of change.

> The "micro" in microservice doesn't refer to the amount of code, but rather the surface area of its responsibilities; a microservice should be focused on one, and only one, domain concept, rather than putting all the services into a single deployment target.

Now the project consists of multiple Docker images, and starting and stopping all of them in sync can be a pain. It's Docker community to the rescue! Several projects, two of which are Kubernetes and Docker Swarm, specifically aid with the management and lifecycle of multiple Docker images in sync, so that the entire system can come up with a single "docker-compose up" command. (Which of those two will "win" in the end is a hot debate right now.)

This wouldn't be a Docker article if the word "microservices" doesn't get mentioned at least once, but there's a reason for that. The single-process nature of Docker encourages smaller deployment targets. The fact that Docker only allows a single process to run means that developers and architects have to figure out what that one process will do, which encourages a smaller, more single-purpose-focused design.

Lastly, using Docker means never having to write a Microsoft Word doc with install instructions ever again. Combined with the fact that every major cloud vendor now supports Docker directly, developers can now make sure that the code that they write will be deployed exactly the way it needs to be in order to run correctly. In essence, Docker allows us to package up the code and its immediate surrounding environment into an image deployable to anywhere, from anywhere, and it will simply work.

And that, my friends, was the point all along.

Ted Neward

Building an Alexa Skill with AWS Lambda

In my previous Alexa article, I covered building an Alexa Skill using Azure. In this article, I'm covering similar territory, but this time, instead of Hello World on Azure, I'll show you how to build a City Facts skill as an AWS Lambda Function. One of the more popular beginner Alexa Skill projects is a Facts Skill. This simple skill is a collection of facts about the subject of your choice.

Chris G. Williams
chrisgwilliams@gmail.com
www.geekswithblogs.net/cwilliams
twitter.com/chrisgwilliams

Chris Williams is a Senior Developer for Level One. He is the author of Professional Windows Phone Game Development (WROX) and numerous articles in CODE Magazine.

He's also a nine-year MVP award recipient, MCT, MCSD, blogger, hardware hacker, and cryptocoin enthusiast.

He's the author and maintainer of the CryptoConnector (open source) project, author and maintainer of Heroic Adventure! plus, a freelance game developer, conference speaker, INETA Community Champion, and former INETA Board of Directors member.

Some popular skills of this type include trivia facts, such as Superman Facts, Batman Facts, etc. (Asking Alexa for "Superhero Facts" will result in some really awful jokes. You've been warned.)

Facts Skills are part of the Custom Skills library. Almost all skills you create will be Custom Skills (aside from Smart Home Skills and Flash Briefing Skills.) The Facts Skill outlined in this article will be full of information about my favorite place in the Marvel Cinematic Universe: Wakanda. (Wakanda is property of Marvel Comics, Disney, etc.)

A Brief Recap on Alexa Skills

If you read my previous Alexa article in the Mar/Apr 2017 issue (http://www.codemag.com/Article/1703061/Programming-Alexa-Skills-for-the-Amazon-Echo), this section should feel familiar.

First, some terminology: Alexa is the personality, and the intermediary voice service between the device and your skill. The devices can come in many forms, including an Echo, Echo Dot, Echo Show, Echo Spot, and Fire TV, to name a few, with more devices coming out all the time. The skill is the code you've written that responds to a request from Alexa.

That's a bit of a mouthful, so take a look at **Figure 1** for a clearer picture of how it all comes together.

When you invoke a skill with Alexa, it requires a structured command, such as: "Alexa, give me a Wakanda Fact" (which, coincidentally, is very similar to the command used to invoke the skill developed later in this article).

When designing an Alexa Skill, you need to ask yourself the following questions:

- What purpose does it serve? (To provide information about Wakanda.)
- Where will it live? (in an AWS Lambda Function)
- How will it be invoked?
- How will it respond?

The first two questions have already been addressed, and the last two are covered in the following section, so keep reading.

Marvel Wakanda Facts

Asking Alexa for a Wakanda Fact currently gets a response about Wauconda, Illinois, so instead this skill will be invoked by asking for a "Marvel Wakanda Fact."

Because not everyone asks for things in exactly the same way, we'll need a few sample utterances so that Alexa can recognize the request:

- Give me a Marvel Wakanda Fact
- Tell me a Marvel Wakanda Fact
- Tell me about Marvel Wakanda
- Tell me something about Marvel Wakanda
- What do you know about Marvel Wakanda?

That should cover the basics, and you can always add more to the list. I'll map these to an **intent** (which is essentially how Alexa refers to the functions your skill offers), which I'll call:

- WakandaFactIntent

And to round it out, I'll throw in some of the built-in intents that Alexa supports:

- AMAZON.HelpIntent
- AMAZON.StopIntent
- AMAZON.CancelIntent

You know where it's going to live (AWS) and you've got the utterances, and the intents, so the design is almost done. All that remains is knowing what will be returned and how. The how part is easy enough, as you'll be returning a JSON message back to Alexa containing a randomly selected fact.

I've made the "what" part easy too, by grabbing a handful of Wakanda facts from Wikipedia. As you can see, there aren't very many here, but you can always add some of your favorites.

- "Wakanda is an African nation appearing in American comic books published by Marvel Comics. It is the most prominent of several native African nations and home to the superhero Black Panther. Wakanda first appeared in Fantastic Four #52 (July 1966), and was created by Stan Lee and Jack Kirby."
- "Due to its intentional isolationism, Wakandan technology has, until recently, developed entirely independently of the rest of the world. As such, the design philosophies and methodologies are different and often incompatible with conventional equipment."
- "Wakanda is the world's most technologically advanced country. For example, Wakandan computer technology is more powerful than that of the rest of the world and completely immune to outside hacking, as it is not based on binary electronics; it can, however, emulate the behavior of such electronics at hugely enhanced efficiencies, allowing it to easily hack almost any conventional system."
- "Vibranium was used liberally in Wakandan technology, but the recent destruction of all Vibranium has forced large-scale redesigns."

With that, the design is essentially done and it's time to create the skill in the next section.

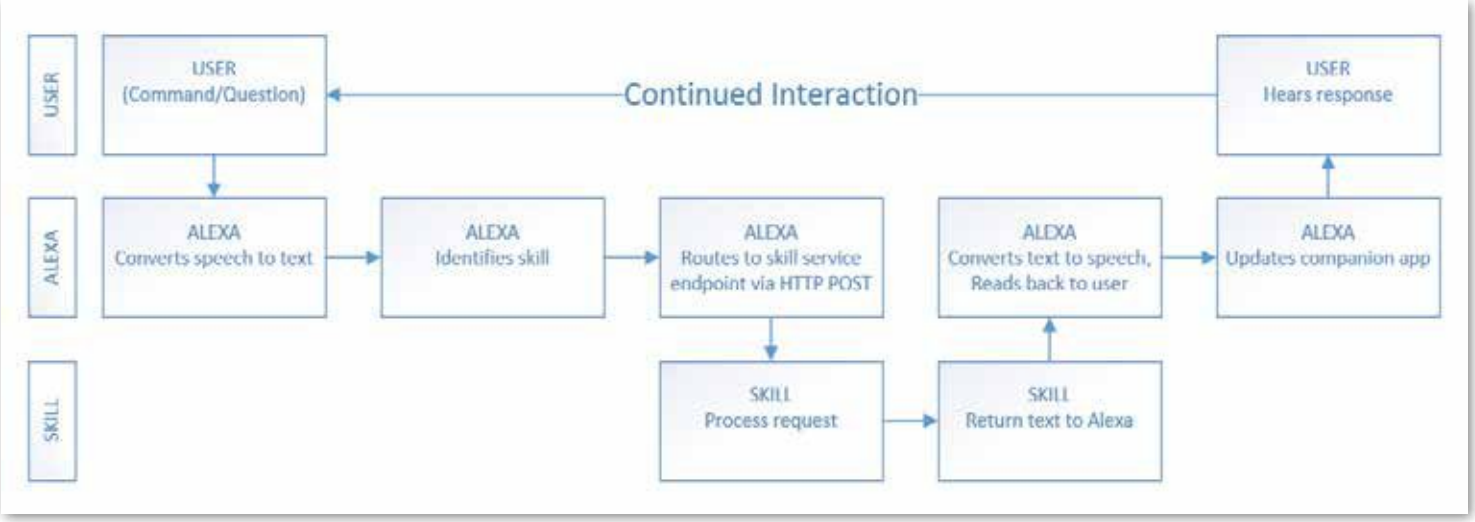


Figure 1: Alexa Skill Voice Interaction

Building the Skill

Because you already have the intents and sample utterances, go ahead and convert them into JSON.

First up, the intent **schema**, as a JSON array:

```
{
  "intents":
  [
    {
      "intent":"WakandaFactIntent"
    },
    {
      "intent":"AMAZON.HelpIntent"
    },
    {
      "intent":"AMAZON.StopIntent"
    },
    {
      "intent":"AMAZON.CancelIntent"
    }
  ]
}
```

Next is the sample utterances mapping, in plain text:

```
WakandaFactIntent give me a marvel wakanda fact
WakandaFactIntent tell me a marvel wakanda fact
WakandaFactIntent tell me about marvel wakanda
WakandaFactIntent tell me something about marvel
 wakanda
WakandaFactIntent
```

Again, feel free to add more if you think of them, just be sure to include the name of the intent first. You don't need to add the built-in intents (i.e., Help, Stop, and Cancel). Alexa is pretty good at interpreting those commands, but if you have a specific phrase you want to map to one of them, go for it.

You can save both of the snippets into a single text file, as you'll just be copying and pasting them into the Amazon Alexa Developer Dashboard in a little bit.

Before heading off to the dashboard, there's still the matter of writing the skill code to return your Wakanda facts.

Node.js Code

I'm using Node.js, as it's one of Amazon's preferred languages for AWS Lambda functions and it shows up in most of their samples. If you don't know it, don't panic. I won't be using enough of it to differentiate it from basic JavaScript.

To start, you need some basic boilerplate code, which includes a reference to the Alexa SDK, and setting up the event handler that will be used to respond to the various intents. You'll use this code practically every time you make an Alexa skill as an AWS Lambda function:

```
'use strict';

const Alexa = require('alexa-sdk');

exports.handler
   = function(event, context, callback)
{
    var alexa = Alexa.handler(event, context);
    alexa.APP_ID = 'amzn1.ask.skill.replace-me!'
    alexa.registerHandlers(handlers);
    alexa.execute();
};
```

The next part is a collection of functions that respond to various Alexa events, such as LaunchRequest, and your intents. The LaunchRequest event is fired automatically whenever your skill starts up (on a new interaction, not only a response).

In this example, I'm emitting the WakandaFactIntent that triggers the function of the same name, and which calls the getRandomWakandaFactEvent that I'll get to shortly.

You can also see handlers for the Cancel, Stop, and Help intents:

```
var handlers = {
    'LaunchRequest': function () {
        this.emit('WakandaFactIntent');
```

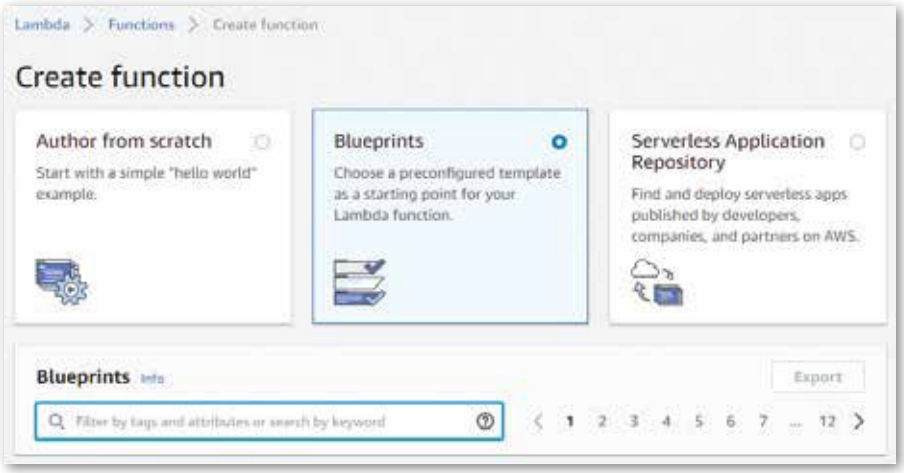


Figure 2: The Create Function dialog

```
    },

    'WakandaFactIntent': function () {
        var say = 'Here is your fact. '
            + getRandomWakandaFact();
        this.response.speak(say);
        this.emit(':responseReady');
    },

    'AMAZON.CancelIntent': function () {
        this.response.speak('Goodbye');
        this.emit(':responseReady');
    },

    'AMAZON.StopIntent': function () {
        this.response.speak('Goodbye');
        this.emit(':responseReady');
    },

    'AMAZON.HelpIntent': function () {
        this.response.speak('you can ask for a
            fact by saying, tell me a fact.');
        this.response.listen('try again');
    }
};
```

Most of this is boilerplate stuff as well, other than the custom event and function names, so feel free to use it over and over to create your own Fact Skills.

Most of this is boilerplate stuff, other than the custom event and function names, so feel free to use it over and over to create your own Fact Skills.

Lastly, there's the getRandomWakandaFact function that selects a fact, at random, and sends back a string for Alexa to say.

In this example, I'm storing all of the facts in an array, but it could just as easily be a database, a call to a web-service, or whatever you need to get the job done. I'm only showing a snippet of it here, but you can see the entire function in **Listing 1**.

```
function getRandomWakandaFact() {
    var myFacts = [
   'Wakanda Fact 1',
   'Wakanda Fact 2',

   ...

   ];

    return(myFacts[Math.floor(Math.random()
       * myFacts.length)]);
}
```

Those facts are all stored and delivered as plain text, but you could also format them with Speech Synthesis Markup Language (SSML) as needed, if you find Alexa stumbling over any of the words. If you need a primer on SSML, you can take a look at the July 2017 issue of CODE Magazine (http://www.codemag.com/Article/1707091/Use-Your-Words-Getting-Familiar-with-Speech-Synthesis-Markup-Language).

Listing 1.

```
function getRandomWakandaFact() {
    var myFacts = [
   'Wakanda is an African nation appearing in
    American comic books published by Marvel
    Comics. It is the most prominent of several
    native African nations and home to the
    superhero Black Panther. Wakanda first
    appeared in Fantastic Four #52 (July 1966),
    and was created by Stan Lee and Jack Kirby.',

   'Due to its intentional isolationism, Wakandan
    technology has, until recently, developed
    entirely independently of that of the rest of
    the world. As such, the design philosophies
    and methodologies are different and often
    incompatible with conventional equipment.',

   'Wakanda is the worlds most technologically
    advanced country. For example, Wakandan
    computer technology is more powerful than that
    of the rest of the world and completely immune
    to outside hacking, as it is not based on
    binary electronics; it can, however, emulate
    the behavior of such electronics at hugely
    enhanced efficiencies, allowing it to easily
    hack almost any conventional system.',

   'Vibranium was used liberally in Wakandan
    technology, but the recent destruction of all
    Vibranium has forced large-scale redesigns.'
   ];

    return(myFacts[Math.floor(Math.random()
       * myFacts.length)]);
}
```

That's all the code, so it's time to create the AWS Lambda function and set up the skill in the Dashboard.

Set Up and Deployment

In this section, I'll briefly cover setting up an AWS Lambda Function, adding the code, and then configuring the skill in the Alexa Developer Dashboard.

Setting Up an AWS Lambda Function

Before setting up your Alexa skill in the dashboard, you need to create your AWS Lambda function by logging into the AWS Services console at https://aws.amazon.com and selecting Lambda. If you don't have an AWS account, you'll need to create one first.

Create a new function (the big orange button in the top right) and you'll be prompted with a choice of authoring from scratch, using a blueprint, or creating a serverless application repository, as in **Figure 2.**

Select the Blueprints option and type Alexa in the filter box to narrow down the choices. You're looking for the one called **alexa-skill-kit-sdk-factskill**. Selecting a blueprint gives you some sample code and also sets up some of the back-end connections that you'd have to do manually otherwise.

Selecting a blueprint gives you some sample code and also sets up some of the back-end connections that you'd have to do manually otherwise.

With the **factskill** blueprint selected, you'll need to enter some basic info about your skill, including setting a permissions role, as shown in **Figure 3.** I chose MarvelWakandaFacts for the function name, but it can be called anything, this is just an internal name.

Role is a little bit trickier. You'll want an AWS Lambda Basic Execute role, but if this is your first time here, you won't have one in the list, so you'll have to create it. Fortunately, AWS does a good job of walking you through the process, once you select the **Create a new role from templates** option.

This isn't my first time creating an AWS Lambda Function to use with an Alexa Skill, and because you can use the same role over and over, no matter how many you create, I'm using the one I created previously.

At the bottom of the page, you'll also see some generated code for your function. Don't worry about trying to edit it right now; just hit the orange Create Function button. You'll get an opportunity to edit it in a bit.

Now that the MarvelWakandaFacts function has been created, you'll be presented with the designer, which is where you add the triggers from the list on the left. This function requires an **Alexa Skills Kit** trigger to function properly, so select that and you'll end up with something that looks like **Figure 4**.

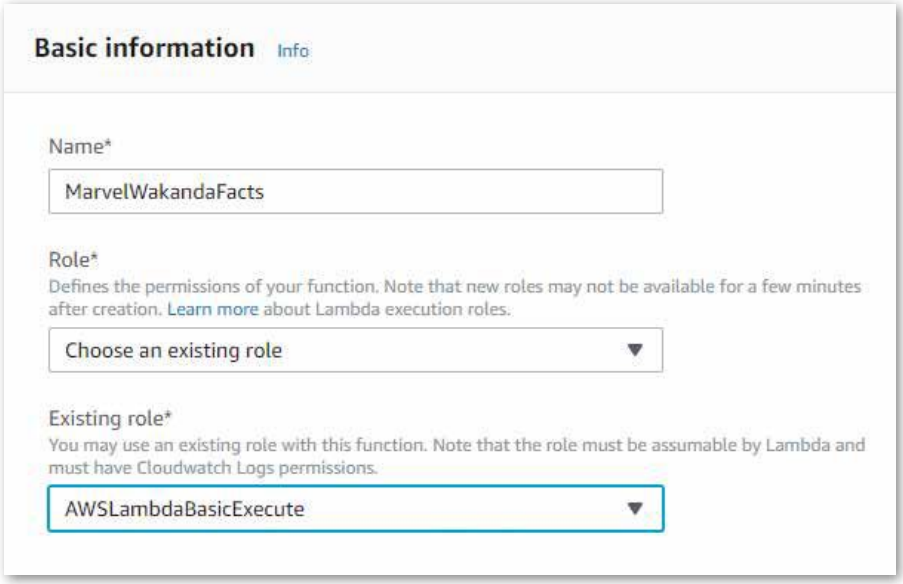


Figure 4: AWS Lambda triggers and resources

Figure 3: The Basic Information dialog box

All of the facts in this example are stored in the code, but if I were building a skill that relied on an AWS-hosted database, it would show up here on the right, under Resources.

Clicking on the Alexa Skills Kit trigger presents a configuration dialog where you can set the Skill ID Verification. This allows you to control which skill(s) can call your AWS Lambda function. For now, just set it to Disable, because you don't yet have a skill ID. AWS isn't a big fan of that, so it's important to go back and change it to Enable once the skill is configured and you have a skill ID.

Save it and click on the MarvelWakandaFacts in the diagram so you can take a look at the code. You'll notice that it's pretty close to what you saw earlier (yay Blueprints!) so it won't require much in the way of changes.

I've made a few format changes to the function code, so it's not a direct match, but it's close enough that you could just paste the facts into the data array (or create your own facts, if you're doing a different topic). I'm going to paste my code in, but I still need an ID, so I'll be back.

Once the code is updated, save again, open a new browser tab (because there will be some back and forth to get everything wired up), and head over to https://developer.amazon.com/alexa to create the skill.

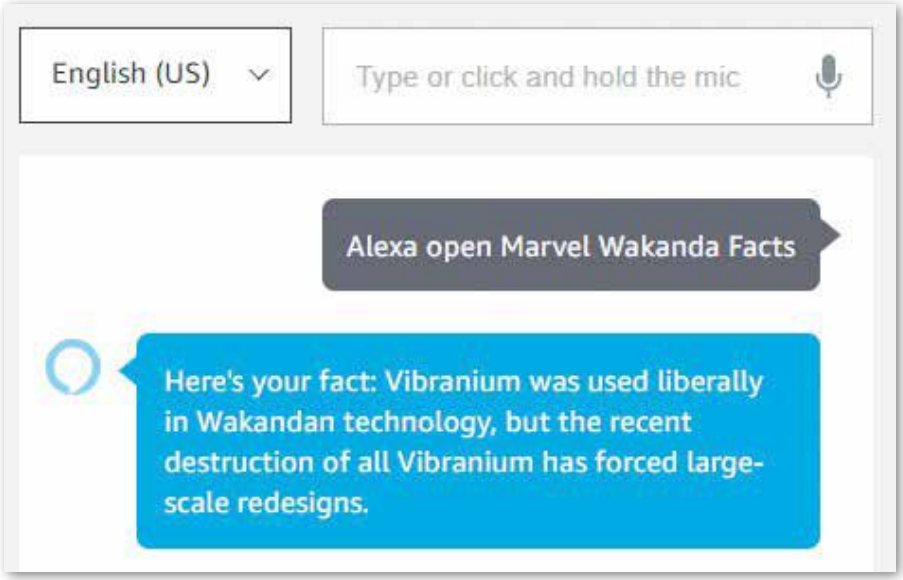


Figure 5: The Alexa Simulator

Creating Your Alexa Custom Skill

In the **Your Alexa Consoles** drop-down, select **Skills**, and you'll be presented with a list of all the Alexa Skills you've created so far, the languages they support, the type of skill, and current status.

Click the Create Skill button and you'll be presented with a wizard to walk you through the creation steps. Enter the skill name, and then select Custom on the next screen.

Once that's done, you'll be presented with the "new and improved" Alexa Custom Skills UI. On the left, you'll see an expandable list that shows your Interaction Model, Interfaces, and Endpoint.

In the Interaction Model section, you'll need to perform the following steps:

1. Click Invocation and enter the Skill Invocation Name. (i.e., Marvel Wakanda Facts).
2. Click Intents. You will see the three built-in intents and still need to add yours. Click the Add button and enter the name of your intent. (i.e., WakandaFactIntent). Make sure what you enter here is an exact match for your intent name, as defined previously.
3. Add the Sample Utterances defined earlier and click Save Model (at the top of the page).
4. Click Endpoint and you'll need the ARN from the AWS Lambda function, so switch back to that tab and grab it from the top right of the page. It looks like this: **arn:aws:lambda:us-east-1:xxxxxxxxxxxx:function:MarvelWakandaFacts**.
5. Copy and paste the entire ARN into the Default Region box under AWS Lambda ARN in the Service Endpoint Type section and click Save Endpoints.
6. Click the Build tab again to refresh the view and take a look at the Skill Builder checklist on the right. Everything should be green except Build Model. Click it and wait a few seconds.
7. Once that lights up green, you're ready to test. Click the Test tab.

Test Your Alexa Custom Skill

You'll need to enable the skill for testing by clicking the slide switch at the top. This automatically validates whether or not your skill's interaction model has been properly built and tells you to go back and fix it if it hasn't.

Once that's done, find the Alexa Simulator tab, and type: **Alexa open Marvel Wakanda Facts**. The simulator will think for a moment, and you'll get a response, like in **Figure 5**.

While you're there, take a look at the JSON Input and Output sections on the right side of the page. These show you exactly what's being passed back and forth between Alexa and your (AWS Lambda Function) code. If you ever run into a problem, these are immensely helpful to look at when debugging.

This works because I disabled checking for the skill ID in the AWS Lambda Function. So that will need to be fixed, and the skill ID should also be added to the function code. You can find the skill ID embedded in the session block of the JSON Input in the Skill I/O section of the Test Page.

It's listed as the applicationId, which I have put in bold, below:

```
"session": {
   "new": true,
   "sessionId": "amzn1.echo-api.session.xxxxx…",
   "application": {
      "applicationId": "amzn1.ask.skill.xxxxx…"
   },
   "user": {
      "userId": "amzn1.ask.account.xxxxx…"
   }
},
```

Copy that, and then go back to your AWS tab and update the skill ID check to Enabled and also paste it into the variable in the code. Make sure to save in both places.

At this point, you're done! Hop back over to the Test screen and try it again, just to make sure everything is still working after you made changes. The only thing left to do now is publish your new AWS Lambda-based Alexa Skill.

Publishing

Publishing is handled via the Launch tab. When you're ready to publish, click the Launch tab and fill out the form on the Store Preview page. You'll need an icon and a description. Once that's filled out, click Submit to Certification and you're done.

Certification usually takes a couple of days, and you'll get an email from Amazon when it's done.

Congratulations! I can't wait to see what you build. Hit me up on Twitter and tell me about it so I can check it out.

Chris G. Williams

Introducing Progressive Web Apps: The No-Store App Solution

For developers, the Web has been in a constant state of change. Starting with static Web pages, developers quickly started building dynamic websites using a combination of server-side- and client-side-rendered content. Arguably, this reached an apex with the advent of Single Page Applications (SPAs). The beauty of building Web applications is how far they can reach.

Chris Love
clove@extremewebworks.com
http://love2dev.com
@ChrisLove

Chris has around 20 years of Web development experience. He has built a wide variety of Web sites and applications. In the past couple of years, he's begun to immerse himself in the mobile Web application space. This is giving him some amazing experiences using cloud technologies, HTML5, and all the major mobile platforms. Currently, he's obsessed with modern Web and mobility to help solve the problems many enterprises have adapting to the rapidly changing technology landscape. He has authored three books, including "High Performance Single Page Web Applications," available here: http://amzn.to/1b0twcm. He's a seven-time ASP.NET MVP, ASP Insider and Internet Explorer User Agent. Chris regularly speaks at user groups, code camps and other developer events. He blogs at http://love2dev.com.

Today, it's a simple process to bring a website online with the capability to reach literally billions of eyeballs.

The downside of "pure" websites is their inability to create a truly rich application that can be encompassed by native applications. That's because 10 years ago, the Web landscape changed radically. What happened 10 years ago? The iPhone happened (along with the Android a short time later). The creation of the iPhone demonstrated to users that portable and rich applications were possible using the Web as a backbone. Although native applications have true advantages over Web applications, they aren't without their disadvantages. They're complex to build, must be built for multiple platforms, are more complex to deploy (because of app stores), and don't have "reach" right out of the box. **Figure 1** contrasts the reach of a traditional website versus the capabilities of a native app on the Web.

The question is: "Can we build applications with both reach across the breadth of the Internet and capabilities similar to native applications?" It's my opinion that the answer to this question is yes. A brand-new way of building Web apps has come to the forefront. This concept is known as a Progressive Web Application (PWA). PWAs combine reach, flexibility, lower development costs, lower deployment costs (no app store hassle), and powerful features that were previously limited to native applications. **Figure 2** demonstrates how PWAs move the bar between capability and reach.

The nice thing about PWAs is that it's not some "pie in the sky" technology. A number of recognizable brands have already jumped on the PWA train. These companies include Starbucks, Lyft, The Weather Channel, The Washington Post, Twitter, and loads of other big names. So how do you jump on the train yourself?

You've come to the right place. Let's take a look at how to build PWAs.

PWA Basics

The promise of PWAs is that they're fast, reliable, secure, engaging, and integrated. There are at least three primary requirements that your website must meet to be considered a PWA. It must be:

- Served via HTTPS
- A valid Web manifest
- A registered service worker with a fetch event handler

These are the big three, although there are many more, and more subtle, requirements to truly take advantage of the PWA spirit. The browser vendors behind PWAs are hoping that we'll soon start delivering Web experiences that implement a battery of best practices.

Service workers provide an extensible mechanism to enable many platform features, like push notifications and background sync. Browsers also trigger Add to Homescreen experiences that make PWAs look and behave like native applications with a first-class presence. (I'll explain more about Add to Homescreen in a minute.)

> "PWAs are a new level of thinking about the quality of your user experience."
> --Chrome's Chris Wilson

Microsoft accepts PWAs to the Windows Store and gives them equal status to Universal Windows Platform (UWP) applications. They're even using the Bing spider data (an SEO crawler) to identify PWAs. If the PWA meets a certain quality level, they automatically submit it to the store.

So far, Microsoft has identified 1.5 million PWAs to include in the store.

> "Homescreen" means just what you think it means: a first screen that leads into the rest of the website's pages.

They've been accepting hosted Web apps to the store since Windows 10 was released in 2015. Up to this point, they were called Hosted Web Apps. They have similar requirements, except for a service worker. Now that Edge supports service workers, Microsoft is changing the name to PWA to sync with the rest of the industry.

Secure by Default

PWAs must be served using HTTPS. This is part of a bigger trend to make the Web secure by default. Many new and existing APIs, like geolocation (see **Figure 3**), Web payments, and log-in forms are being gated behind HTTPS.

Service workers and HTTP/2 require TLS certificates before they are enabled.

Browsers are getting more aggressive with visual cues to users if a site isn't secure. For example, most browsers now detect credit card numbers and password fields. If present, they make it very apparent to the user if the site isn't served via HTTPS (see **Figure 4**).

Browsers are getting more aggressive in visually indicating to users that a website isn't secure. This summer, Chrome 68 will display a "not secure" message in the address bar for any site using HTTP. In the near future, any website served via HTTP and Firefox will show a visible insecure warning to the consumer. Each browser has a road map to increase insecure warning visibility. I advise you to upgrade now, rather than wait, as this alone will severely limit your ability to function online in the near future.

Search engines also use HTTP as a ranking signal HTTPS as a positive ranking signal. Combining search engine ranking with on-screen warnings causes business stakeholders to aggressively upgrade external and internal sites to HTTPS. Today, over 60% of the top 100,000 websites are served via HTTPS and the number grows every day.

If you don't believe me, look at the top 100 search results the next time you perform a search. It's becoming more unusual to see a page served via HTTP.

In the past, HTTPS has been an unfortunate burden for the Web. SSL certificates have been cost prohibitive and a technical challenge for the masses. The vast majority of websites are hosted on a shared server, limiting how much control site owners have over managing TLS certificates. If HTTPS was supported, it's been a premium upgrade, often more than the cost of the content hosting. See **Figure 5** for how things have changed in only the last three years.

Those barriers have been removed in recent years. Let's Encrypt (https://letsencrypt.org) alone has issued over 60 million free SSL certificates. And cloud service providers like AWS and Azure offer free SSL to their CDN customers.

The technical barriers are also being removed. When I add an AWS SSL certificate to a website, it takes me about 30 seconds, and most of that time is waiting on an e-mail to confirm the request.

A big reason behind the rise in HTTPS is that WordPress.com converted all of their customers to HTTPS over a year ago. This was a big chunk of the Web. This forced migration ensured that their customer's sites can take advantage of all the platform features the Web offers while providing a safe experience to visitors.

Using HTTP/2 for Performance

If you're worried about HTTPS causing performance issues, you can stop. HTTP/2 is the latest version of HTTP, correcting many performance limitations of HTTP/1.1. We have used HTTP 1.1 for almost two decades with only minor enhancements. To counter many inherent performance issues, we created many performance hacks, like bundling.

Figure 1: Historically, the Web excelled at reach and native apps excelled with capabilities

Figure 2: PWAs enhance the Web capabilities without losing reach, making them more powerful than native apps

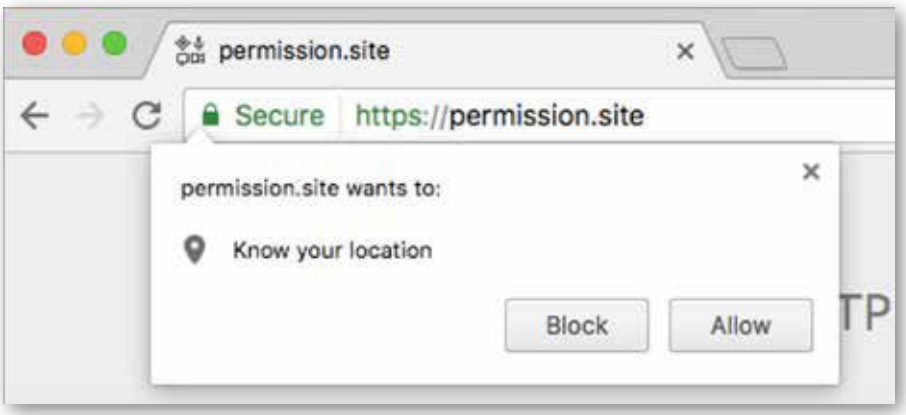


Figure 3: Here's a geolocation request.

Figure 4: You're warned if the site you're about to tell all your secrets to isn't secure.

Figure 5: You can see what happened when things changed to more secure websites.

HTTP/2 features request multiplexing, header compression, and much more. Many of the performance hacks, like domain sharding and bundling are now anti-patterns. HTTP/2 eliminated the need for these hacks by focusing on optimizing file and packet delivery.

Like service workers, HTTP/2 requires HTTPS. The good news is that any good CDN provider has HTTP/2 enabled by default and most make TLS easy to implement. This means that everyone should have support for HTTP/2.

The Weather Channel upgraded to HTTP/2 and saw a significant increase in their Web performance metrics. The increased performance increased their consumer engagement and made their content more consumable on mobile devices.

> "When we launched [HTTPS], we saw an average of a 50ms hit for negotiation… it was more than offset when we activated HTTP/2… a month later we saw a 250ms drop per pageview."
> --Weather.com

The Add to Homescreen Experience

PWAs are designed to be installable. To match native apps on the mobile platform, browsers are implementing an Add to Homescreen experience.

Chrome, FireFox, Opera, and Samsung Internet all offer similar Add to Homescreen prompts. When a user opens your PWA, after a period of time or possibly a repeat visit, the browser may automatically prompt them to add the site to the home screen. They can also manually install a PWA that creates the home screen icon and launch experience.

Figure 6: The Add to Home Screen button makes it easy for users to get to your PWA.

When a PWA is installed using Chrome on Android, it's automatically converted to a WebAPK. This process turns the PWA into a native application on Android. Under the hood, what Chrome does is something similar to a hybrid or Cordova application. At this time, you don't get access to Android-specific platform APIs, but the user can manage the PWA just like any other app.

This is done because Google wanted to avoid writing parallel plumbing to enable PWAs to have a similar experience to native apps. Now your PWA is displayed in the app drawer on Android and shows up like a regular application in the application list. Installed PWAs are included in the Android application settings. This makes it manageable, just like any other native application.

Qualified, Professional Staffing When You Need It

CODE Staffing provides professional software developers to augment your development team, using your technological requirements and goals. Whether on-site or remote, we match our extensive network of developers to your specific requirements. With CODE Staffing, you not only get the resources you need, you also gain a direct pipeline to our entire team of CODE experts for questions and advice. Trust our proven vetting process, and let us put our CODE Developer Network to work, for you!

Contact CODE Staffing today for your free Needs Analysis.

Helping Companies Build Better Software Since 1993

www.codemag.com/staffing
832-717-4445 ext. 9 ● info@codemag.com

CODE STAFFING

You can perform the same process on the desktop. Of course, it's more of an Add to Desktop experience. At this time, no one is offering an Add to Start Menu option. But that should change in the future. (See **Figure 6**.)

The Add to Homescreen functionality is driven by the Web manifest file. This is a JSON document that you host on your Web server and reference from your Webpages. It provides metadata to the mobile device and browser that describes your brand's experience.

The primary fields are **name**, a short **description**, and an array of **icons**. But you can also control how your application will launch once it's been added to the home screen. For example, if you wanted to launch without any of the browser accoutrements like an address bar, you can tell it to do so.

Another important property you define is the **Start URL**. This is a key feature that differentiates a PWA from merely bookmarking a site to the desktop. When you bookmark a site, it creates a shortcut to the current URL. When a PWA is installed, it opens to the designated start or home page defined in the manifest file.

The following code snippet is an example Web manifest file. It's a simple JSON object with a set of properties that describe the PWA to the platform. This example is a minimal manifest file; there are additional optional properties that you may also want to include.

```
{
  "name": "Fast Furniture",
  "short_name": "Furniture",
  "icons": [{
     "src": "/meta/24d5b086-693c-
            a559-1926-8fa98f0b5684.
            webPlatform.png",
     "sizes": "24x24",
     "type": "image/png"
   },...  ],
  "description": "Fast Furniture
       is a Progressive Web
       Application
       Demonstration Site",
  "orientation": "portrait-primary",
  "start_url":
       "/?utm_source=homescreen",
  "display": "fullscreen",
  "background_color": "#F99D36",
  "theme_color": "#55B4F6",
  "scope": "/"
}
```

You can also limit the display mode to landscape or portrait, color scheme, language, and serval other properties (see **Figure 7**). There's an official manifest standard, but browsers may also support custom fields. For example, Windows uses extra properties to describe your application to the store.

If you want to launch with all the browser features, you can do that as well. This means that your progressive Web app can launch from the home screen and take up the entire viewport of the consumer's device, just like a native application.

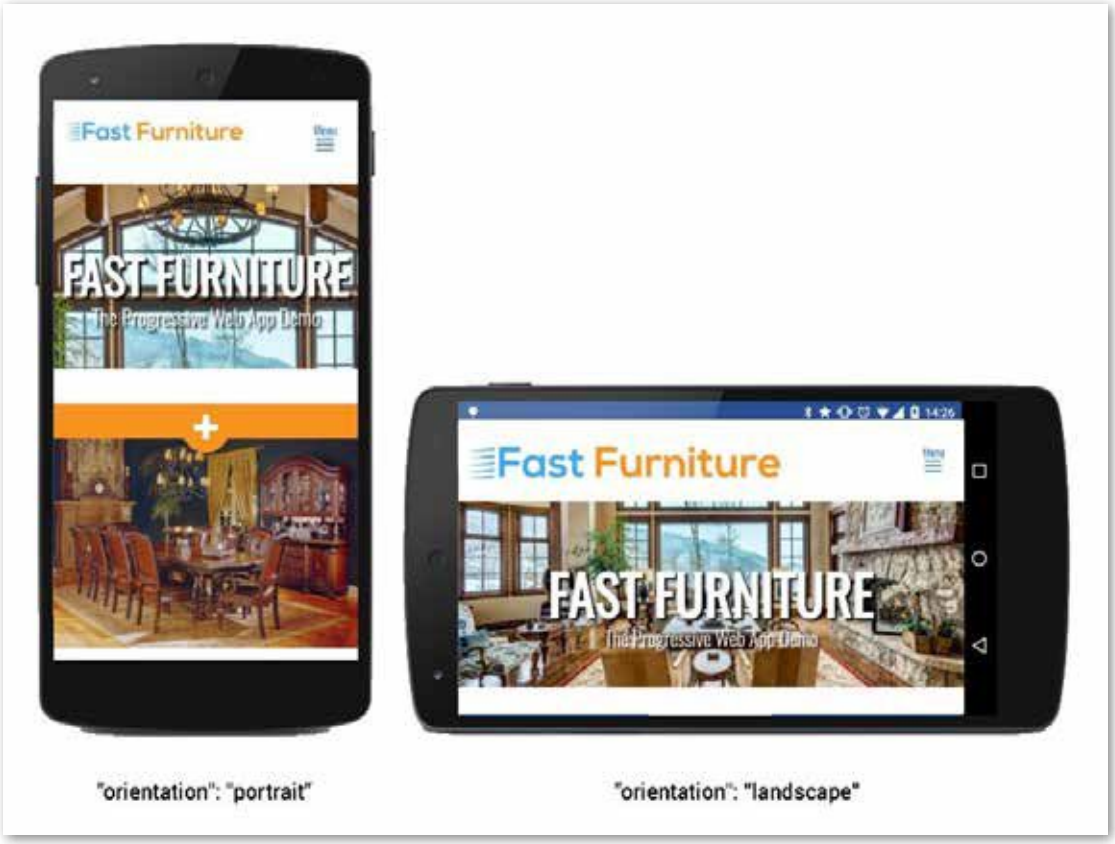


Figure 7: You can orient your home page to best suit your page.

Figure 8: Simple service worker life cycle diagram

Service Workers

Service workers are a brand-new addition to the Web platform. They were first introduced about three years ago by the Google Chrome team. Since then, all major browsers have shipped support for service workers at least in early preview additions.

Service workers are a JavaScript, registered by a Web page, that runs in the background. They can intercept requests to the network, interact with the Cache API, and manage native push notifications and background synchronization. They execute in a separate thread from a browser tab or client.

Service workers are asynchronous, which means they require the use of promises, and activate in response to events. An event can be a tab loading a page within the service worker's scope, a push notification or background sync event. They are designed to be an extensible platform, which means that more functionality can be added later.

Recently, Microsoft and Apple both shipped preview editions of Edge and Safari with service worker support turned on by default (https://love2dev.com/blog/apple-and-microsoft-ship-service-worker-support-in-preview-builds/). Once these two browsers ship support, all major browsers will have service worker support. I don't have exact dates at this time, but I suspect that by autumn, we'll have ubiquitous platform support.

Service workers must be registered from a Web page. You should place the registration within a feature-detection block. This prohibits the service worker registration code from executing in older browsers, avoiding some unwanted exceptions.

```
if ('serviceWorker' in navigator) {
     navigator.serviceWorker
        .register('/sw.js')
        .then(function(registration) {
          // Registration was successful
    })
    .catch(function(err) {
      // registration failed :(
    });
 }
```

You can only register a single service worker per origin (domain). The origin is also known as the scope. A service worker can only control pages and "clients" within the origin that registered the worker. The service worker file must also reside within the scope.

A site could have multiple service workers where they each manage a scope, partitioned by folder structure. For example, the top-level site (example.com) could have a service worker registered. The finance department (example.com/finance) could have a separate service worker. The finance service worker can't access the site's top level and the top-level worker cant access the finance department scope.

The top-level service worker could control the finance department content if it didn't have its own service worker. A service worker's scope can't creep outside of its origin. This makes them secure and prevents bad third-party script providers from injecting themselves into your application.

In addition to Web pages, a service worker client could be any process that initiates a service worker. This includes push notifications, background sync, and other features currently going through the standardization process.

> Each client can only have a single service worker.

A service worker can control multiple clients, like browser tabs, at the same time. Each client can only have a single service worker.

Confused?

I hope not. These concepts belong to service worker life cycle management. This can be a tricky and often overlooked aspect of service workers.

Besides scope, life cycle refers to how service workers are registered, updated and removed. By default, a new service worker can't become active until all existing clients are closed. You can also programmatically force a new service worker to take control by calling the skipWaiting method in the install event. (See **Figure 8**.)

When a service worker is registered, there are a couple of events to which you can bind handlers: install and activate. When the service worker is registered, the install event triggers. When it becomes active, the activate event triggers.

```
self.addEventListener('install',
  function (e) {

    //do something

});

self.addEventListener('activate',
  function (event) {
```

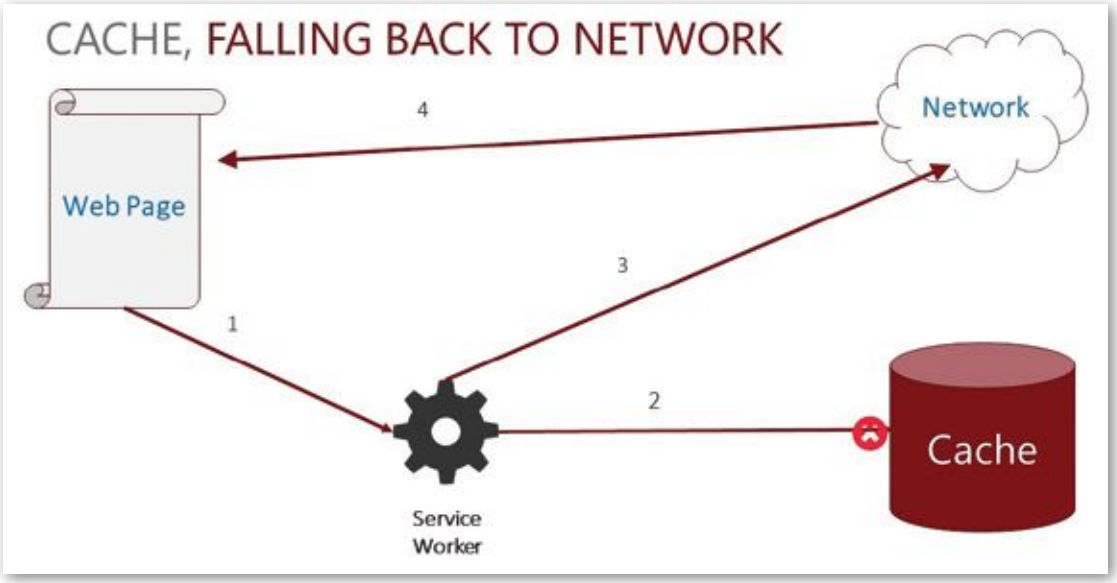


Figure 9: Caching your most popular pages and assets can reduce load time for your users.

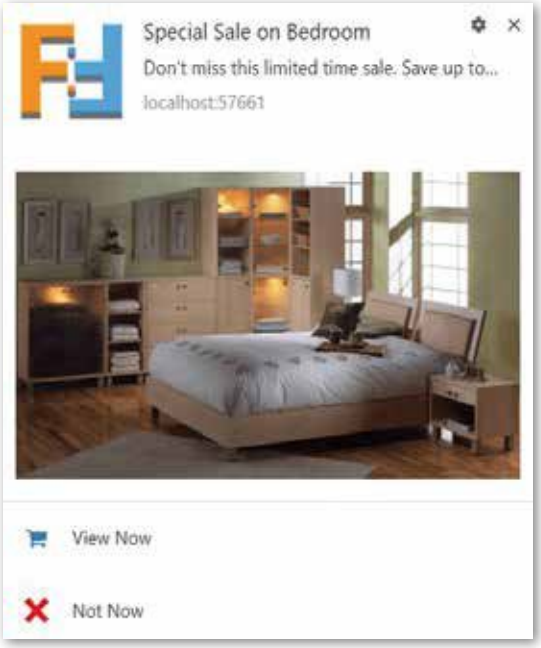


Figure 10: The cache falling back to the network strategy

```
    //do something
});
```

Don't discount the importance of managing your service worker's life cycle. These events give you the ability to perform tasks to set up your service worker. The most common tasks are pre-caching network resources and cleaning named caches when the schema changes.

Service workers run in an external process or thread from the browser. They can intercept every request to the network, allowing you to control the response. They also enable other features, like push notifications, even when the browser isn't running.

Part of the service worker specification is a robust cache infrastructure. The service worker cache gives you the ability to persist network responses in the browser and return responses from cache instead of hitting the network. By caching resources locally, you make the network an optional requirement. Now, there's no network latency when you request a resource.

This feature alone enables rich off-line and instant loading experiences. The cache API makes the network an enhancement and not a requirement.

How Service Worker Caching Works

Service workers can intercept every request to the network and determine how that request will be handled. It does this through the fetch event handler.

The reason why it happens in the fetch event is that the service worker API depends on the Fetch API being implemented in the browser (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch). If you haven't heard of the Fetch API, you should take some time to learn how it works. It's a modern replacement for XMLHttpRequest, or, as we commonly know it, Ajax.

Ajax is a completely rewritten implementation that leans on promises and a much simpler API. It also includes custom Response, Request, Header, and Body objects that you can manipulate to craft custom versions as needed. The Cache API depends on these fetch objects to persist network responses across sessions.

The fetch event handler allows you to create your logic workflow to determine how requests will be handled. The fetch event has an Event object that contains a reference to the Request. By passing the Request object to the Cache API, you can determine if a response is already cached for that particular request. If so, you have the option of returning the cached response or hitting the network, or, if you are so inclined, doing both. This means that you have complete control over how your application is persisted in the browser.

Listing 1: Clone the response and store it in cache

```
self.addEventListener("fetch", event => {

    event.respondWith(
        caches.match(event.request)
            .then(function (response) {

                if (response) {
                    return response;
                }

                return fetch(event.request)
                    .then(response => {

                        if (response.ok ||
                           response.status === 0) {

                            let copy = response.clone();

                            //if it was not in the cache
                            //it must be added to the
                            //dynamic cache
                            caches.open(dynamicCache)
                                .then(cache => {
                                    cache.put(event.request, copy);
                                });

                        }

                        return response;

                    });
            })
    );

});
```

Asset Pre-Caching

When you first install a service worker, there are several events that you can use, one of which is the install event. Here, you can perform many different tasks. A very common one is to precache common assets.

The following code snippet fetches a list of important URLs from the network and places them in a named cache. These assets typically belong to the app shell (the common site layout) or are identified as frequently needed or above the fold resources. This list contains common HTML, JavaScript, CSS, fonts, and images.

By caching them when the service worker is installed, you're ensuring these resources are immediately available when the app is loaded.

```
self.addEventListener('install', function (e) {

  e.waitUntil(
    caches.open(cacheName).then(function (cache) {

      return cache.addAll(filesToCache)
      .catch(function (error) {
              console.log(error);
      });
    })
  );
});
```

This is what is called pre-caching and you need to select files that are used frequently, like your main site's CSS and JavaScript files, common images, and popular pages. Pre-caching these files insures that they're available whenever someone requests them. This means that those pages and those assets can load almost instantly, without having to download from the network (see **Figure 9**).

This is a huge advantage because many of the files I mentioned are commonly accessed across your website. Because they're stored locally, you no longer need to hit the network. This is a valuable feature for traditional desktops on broadband as well as mobile devices on cellular networks. The fastest request you can make is the one that never happens, or at least that's the way the saying goes. Accessing responses from cache has negligible latency, measured in a few milliseconds. Often, my experience is that the latency is much less than a single frame-refresh on the display.

Cache First Falling Back to the Network

Let's look at a common caching pattern, the cache first and network fallback strategy. This is one of my favorite caching patterns and allows me to return network assets instantly to the client (see **Figure 10**).

This pattern requests a resource and the service worker intercepts the request. It then checks to see if there's a cached response. If there is a cached response, it's returned and the network access is never accessed.

If a response isn't currently cached, the request is passed along to the network just as if the service worker didn't exist. When the response is received, the service worker can then clone the response and store it in cache for later access (see **Listing 1**).

There are many caching patterns that you can employ in your application. I like to say that it depends on your application and data's personality. Personally, I pull from over 20 different patterns, but they generally use the cache first, falling back to the network strategy as their base.

You can also cache when the device is offline and provide a fallback. For example, you can add service worker templating to your caching strategy and create a custom response on the fly. You could also provide generic fallbacks for different content types.

You May Be Wondering About Traditional Browser Cache?

Traditional browser cache doesn't allow you to have any control over how and if network assets are persisted. Cache control headers give you some knowledge over how the assets may be stored, but browsers can purge as they feel necessary.

This is a been extremely problematic on mobile devices as browser cache is purged very aggressively, meaning your website or application can have much more network

chatter than you ever anticipated. This is one of the many reasons why the average webpage today has a more then 19-second load cycle.

More Than Just A Cache Machine

Caching isn't the only thing that service workers provide. Because service workers function in an external process and are event driven, they can respond to external stimulus.

The first exciting feature to take advantage of this is native **push** notifications. This has been the last big native application feature that the Web has been lacking. It's one of the few native features potential customers have asked me to implement on the Web and I simply couldn't offer.

Service Workers Provide a Native Push Notification Experience

We've had the ability to do notifications through Web workers for a few years. The drawback is that those notifications require the browser to be open. Now, push notifications can work even if the user isn't actively using their device. This gives businesses the ability to engage with and interact with customers anytime. For example, if I have a new blog post, I could send a push notification to all my push notification subscribers that a new post has been made. Statistically, this means that my readership will go up.

If I'm a retailer, I can promote specials and new products that the customer may find interesting, as shown in **Figure 11**.

There are additional service worker features in the pipeline. Browsers are already shipping background sync, for instance. Other capabilities are being debated and we should start seeing implementations soon. Service workers are designed to be extensible and serve as a new platform for the Web.

Service Workers: The Serverless Browser Platform

I think the best description I've heard of service workers is that it's a proxy server in the browser. Although it's not technically in the browser, it is commonly associated with the browser.

> Service workers briefly perform a small task and then go quietly away.

I think a good analogy for service workers is that of a cloud-based function like AWS Lambda or Azure functions. If you're not familiar with them, they're tiny task-specific features that you can implement in any cloud platform that spin up for a very short period of time do the one task they need to do and then quietly go away. They don't consume expensive computer resources and in the case of our mobile phones, service workers won't drain the battery.

Performance as a Requirement

So far, I've discussed technical changes to the Web to enable PWAs. But there's more to it than just some of these

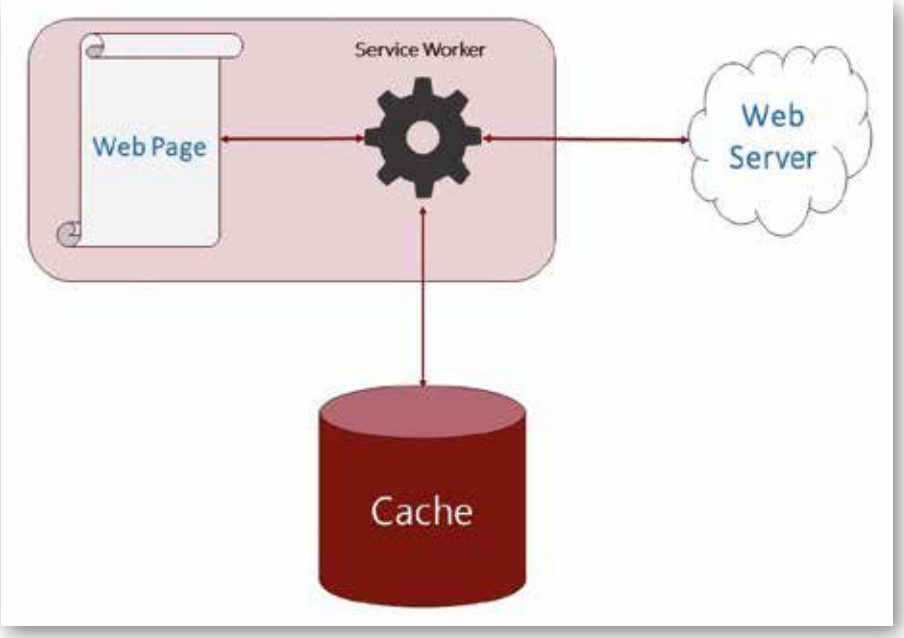


Figure 11: Push notifications keep your customers engaged.

new platform features. PWAs are about delivering the best user experience possible through the Web. And that means that you have to implement Web development and architectural best practices.

Unfortunately, this doesn't always happen. The 19-second average page load time that the Web currently suffers from is largely due to poor development practices. The Chrome team has been promoting best practices for the last four years, known as RAIL and PRPL. (See the sidebar for a brief definition of RAIL and PRPL.)

These two patterns are all about designing your code to work as efficiently as possible in the browser. When doing so, you eliminate almost all the scenarios they create that are what we might call "janky." This is where the page re-renders itself or jumps around as the user's trying to read it. And that's because the JavaScript is typically out of control and manipulating the DOM and in an uncontrollable manner.

Being Progressive

PWAs are simply existing websites that implement these new features as they become available. This is where the progressive part comes into play. When you have a progressively enhanced website, modern features and functionalities only kick in when the platform (browser) supports them.

For example, you don't want to try to register a service worker in a browser that doesn't support service workers because it throws an exception. But if you do feature detection, you can determine if service workers are supported. This means that you can dynamically load the script to execute those new features based on platform support.

If a feature has a polyfill, a JavaScript fallback, you can also feature-detect to determine if the polyfill should be

loaded. I always check to see if Promises and the Fetch API are supported. If one of them isn't, I load polyfills to add the missing feature.

You can do this for many modern features, and dynamically load polyfills as needed. This means that your page and site's loading profile will be as lean as possible based on the user's browser.

Which Browsers Support PWAs?

As of the end of 2017, we have a very good browser support story, as far as PWAs are concerned. Microsoft Edge and Apple Safari are the last two browsers to implement full PWA functionality. All browsers now support HTTPS and HTTP/2.

Apple Safari is the only one that hasn't shipped a Web manifest implementation. Apple began work on it this past autumn and recently shipped support in a technology preview.

With Microsoft and Apple on the PWA bandwagon, we have a full set of platforms engaged. This means that you can expect anyone anywhere to experience your PWA.

The Future of Frameworks and Single Page Apps

A common question many have is how PWAs relate to single page apps (SPAs). A SPA can be a PWA, but a PWA doesn't need to be a SPA. In fact, I'd argue that PWAs will bring SPAs as we know them to an end.

By removing that overhead from the UI thread and moving it to the server to pre-render you can sit back and watch load times improve. This also means that you don't need to deliver as much code to the client.

This has a big impact because your site's payload is much smaller than without PWAs and that you're moving work previously done on the UI thread to the background or the server. Now you can leave the UI thread to do what it's really good at doing: rendering the DOM.

I think the days of heavy SPAs are numbered because we no longer need to worry about page transition delays. In the past, page load latencies created an unwanted delay as a user navigated from page to page. Because your site assets should be cached using the service worker, pages can load instantly, with no network latency.

This doesn't mean that your entire site will be pre-cached, but you should use caching strategies that provide a rich experience. You may still employ some SPA techniques, like an app shell you "fill in" once markup is received from the server. This is where advanced caching and rendering strategies can be employed.

Service workers can be very simple or very complex. It's up to you to determine how much you're going to learn in order to implement service workers. If you want to be a programmer on the Web, it's my advice to learn how to use service workers properly and understand the power and functionality that they offer. I'd also advise you to look at how the server is used in your PWA. Just like modern JavaScript dependencies are changing, so are Web server requirements and functionality.

The good news is that you can upgrade existing websites to a PWA without changing existing code. If your application is a SPA, it'll still work just fine. You can also make your site work off-line and load instantly.

You don't have to change the way you're laying out your application because a mobile-first responsive website is exactly what a PWA should be. If you're rendering everything on-demand on the server, you can continue to do so and your PWA will work just fine.

As you add PWA functionality, I guarantee that you'll start seeing some of the benefits I've discussed. You can start doing much of the work that you been putting onto the client-side and the server into the service worker. You'll see that it can be done much more efficiently there.

PWAs offer a brand-new experience for the Web that should engage customers and end-users more than ever before. They are a force that's going to make the Web equal to or better than native applications. With the decline of native applications, it's time for the Web to step up and take its place as the preferred client platform again. Just like we did on the desktop some 10 years ago.

Chris Love

CODE

RAIL and PRPL

RAIL breaks down the user experience into key actions. Using RAIL principles helps target development for the best impact on users interacting with your app. RAIL stands for **Responsive**, **Animated**, **Idle**, **Load**.

PRPL is a pattern for structuring and serving PWAs to improve launch and app delivery. It stands for **Push**, **Render**, **Pre-cache**, **Lazy-load**.

(Continued from 74)

from their scheduled sprints to do some of the work necessary to stand up the pipeline? It's time to go to the customer and pitch the idea their way. It turns out that they're not opposed, so long as nothing changes the schedule. Well, obviously some level of schedule impact is likely, because standing up the pipeline is probably going to bump into a problem here or there that will require some fixing—so now you have to explain to the customer that although there might be a short-term schedule adjustment, in the long-term, the schedule might actually benefit, and what's more, quality should go up as a result.

And so on, and so on, and so on. The idea is still the same, but whereas the developer is responsible for executing at the keyboard, the manager is responsible for executing in the meeting room. And if you're a developer, thinking about moving up the ladder and facing that reality, it can very easily turn into an internal exercise in constant Imposter Syndrome: "I have no idea what I am doing, and I am desperately trying to keep anybody from figuring that out."

Managing to Improve

But think about your own past for a moment—when you first started out in the industry as a developer, you had no idea what you were doing then, either. (Neither did I.) How did you get to the point of feeling at least even a little bit comfortable? Likely as not, you did one (or more) of three things:

- Learned from role models. There were developers on the team whom you consciously emulated. You watched them, you learned how they made decisions, and you took their advice (or at least as much of it as made sense to you at the time).
- Set goals to learn more. Yes, you probably had some performance goals, but most of us in the early days also had some goals about learning. Sometimes they were pretty vague ("I want to learn as much as I can"), or sometimes they were pretty specific ("I want to learn how to write a game"), but even if they weren't phrased as "learning" goals, you had goals that required learning.
- Understood that your "true self" was changing. In the beginning, you may have felt like claiming the title developer was something of a fraud, but over time, you came to accept and ingest it as a part of your sense of self. You let the story of you change over time.

Does any of this sound like you're somehow hiding or not being true to your sense of self? There's an accusation waiting just under the surface here, that I'm trying to tell you to "be somebody you're not".... But then again, isn't that exactly what education and training and growth is?

A few years ago, on a lark, I took a few classes in glassblowing. Before those classes, I'd never really thought about glassblowing or becoming a 'blower, but after the classes, I can call myself a glassblower—a pretty terrible one, mind you, but a glassblower just the same. I was literally trying to "become something I wasn't" by taking the class. And the same would be true of whatever I decide I want to try to do tomorrow—bass guitarist, cook, speaker of Russian, or CTO of a Fortune 1000 company.

All of which leads me to say, "To thine own desired self, be true." Unless it's to write Perl, of course—then, you should just take up knitting. It's safer for all of us that way.

Ted Neward

CODE

May/Jun 2018
Volume 19 Issue 3

Group Publisher
Markus Egger

Associate Publisher
Rick Strahl

Editor-in-Chief
Rod Paddock

Managing Editor
Ellen Whitney

Content Editor
Melanie Spiller

Writers In This Issue
Kevin S. Goff
Chris Love
Ted Neward
Paul D. Sheriff
Wei-Meng Lee
Sahil Malik
John V. Petersen
Chris Williams

Technical Reviewers
Markus Egger
Rod Paddock

Production
Franz Wimmer
King Laurin GmbH
39057 St. Michael/Eppan, Italy

Printing
Fry Communications, Inc.
800 West Church Rd.
Mechanicsburg, PA 17055

Advertising Sales
Tammy Ferguson
832-717-4445 ext 26
tammy@codemag.com

Circulation & Distribution
General Circulation: EPS Software Corp.
International Bonded Couriers (IBC)
Newsstand: Ingram Periodicals, Inc.
Media Solutions

Subscriptions
Subscription Manager
Colleen Cade
ccade@codemag.com

US subscriptions are US $29.99 for one year. Subscriptions outside the US are US $44.99. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa, and Discover credit cards accepted. Bill me option is available only for US subscriptions. Back issues are available. For subscription information, e-mail subscriptions@codemag.com.

Subscribe online at
www.codemag.com

CODE Developer Magazine
6605 Cypresswood Drive, Ste 300, Spring, Texas 77379
Phone: 832-717-4445
Fax: 832-717-4460

On Authenticity

For an industry that prides itself on its analytical ability and abstract mental processing, we often don't do a great job applying that mental skill to the most important element of the programmer's tool chest—that is, ourselves. "To thine own self, be true." (Socrates) "Authenticity is about being true to who you are." (Michael Jordan) "You gotta be you." (Popular saying) "Be yourself, everyone else is taken." (Oscar Wilde)

The quotes are legion, and the sentiment, endless: No matter what else, it's clear that in the 21st century, it's a given that we should aspire to that state of being in which we are most genuinely "ourselves." No pretending, no "passing," no faking it; just genuine 100% authenticity.

Would it surprise you that this is actually a less-than-desirable state of affairs, particularly for those in management?

Who Are You, Really?

Let's start from the basic premise on which authenticity rests: that, deep down, you have a "true self," and that you can either spend time trying to deny that "true self," time and energy that will distract you from whatever other struggles take place in your life. That the discordance of trying to be something you're not will eventually overwhelm and strangle you, and it's far easier to simply embrace that "inner you" and let it out and let the others who would criticize that be damned.

What if your true self is a serial killer?

Let me admit something: I don't believe I have a true self. Or, to be more precise, I don't believe there's an "inner me" looking to struggle its way out. (You may feel differently, and that's between you and the mirror.) To be more precise about it, I disagree not with the "inner" part, but the "me" part; I don't believe that the sense of "me" is entirely singular or even finished. Ten years ago, professionally, that "me" was a speaker, consultant, and architect. Twenty years ago, that me was a C++ developer who was firmly convinced that nobody around him could see that he was smarter than he was given credit for. Five years ago, professionally, that me was a speaker wrestling with the loneliness that comes from being on the road too much and growing estranged from his family.

And all of that only speaks to the professional parts of me; as a father and a husband, that me has changed drastically, particularly as my boys have grown into men. My responsibilities as father are different now. Where before, I was responsible for teaching them responsibility, ethics, and morals, now my job is simply to teach them how to be a decent roommate. (At 24 and 18, they're not kids anymore.) As a man, that me has also changed drastically. Where I used to count "swimmer" and "soccer referee" as among the core parts of me, that's not really there anymore. I mean, sure, I haven't forgotten how to kick the ball, but where the mind is willing, the body is...let's just say it's not quite as willing.

All of this basically underscores the larger point that yes, although it's important to be authentic, it's also important to understand that you, me, and all of the rest of humanity, are all a work-in-progress. Even if you count yourself without flaw (which is a red flag in and of itself, by the way), experiences and exposure to new people or ideas can change the way you look at the world, which, in turn, can change the way you think about yourself. More dangerously, it's often easy to hold on to the idea of "authenticity" as an excuse to hold on to behaviors and ideas that feel comfortable, when in fact they're doing damage than good.

Managing to Get By

In the Harvard Business Review's "For New Managers," the essay "The Authenticity Paradox," by Herminia Ibarra, talks about how new managers can often see authenticity as an "unwavering sense of self," which can lead to a struggle to take on new challenges and bigger roles. This is true of all of us, to be honest. After all, if your "unwavering sense of self" says that you are a software developer, how can you possibly consider being "untrue to yourself" and taking on a team lead role? Developers tell computers what to do, not people.

So when offered the opportunity to take on a lead position, the "true-to-self" individual faces an uncomfortable catch-22: agree to take on the new challenge and role while feeling that this is somehow beyond them or un-true to themselves, or they can refuse the opportunity and watch somebody else take the reins and always wonder "What if?" particularly when situations arise in which the thought is, "Wow, I would've handled that differently." But refusing the opportunity has a deeper penalty, in that developers who turn away opportunities to climb the ladder are too-easily dismissed from future opportunities, including ones that might feel more comfortable. "If George wasn't willing the take the risk of being a team lead," the executive committee thinks, "Why would he be open to becoming an architect on the upcoming project?" George might well be able to articulate why the difference, but he'll never be given the chance to do so, and probably won't even know that the opportunity was a consideration.

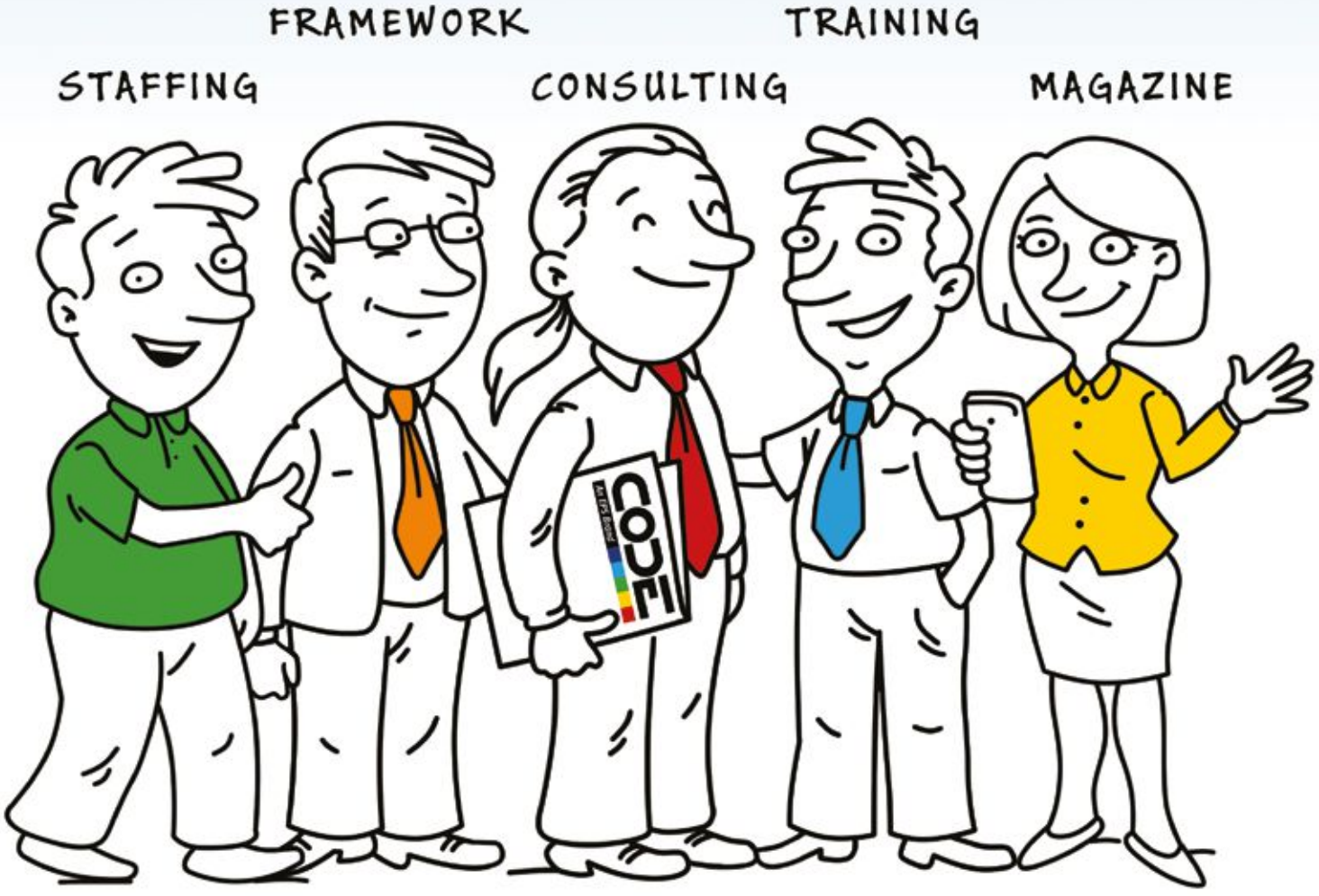
George managed to stay inside his comfort zone, to be sure. He stayed true to his sense of self. But he also got pigeonholed as a developer, and once in the pigeonhole, it's ridiculously difficult to climb back out of it. And the longer George stays in that hole, the deeper and slipperier it gets.

For managers and leaders, particularly those who are new to the role, part of what compounds the feeling of dishonesty-to-self is the fact that the nature of the work shifts entirely: from thinking to talking. As a developer, we spend our time thinking about features, or bugs, or architecture, and then how to design it, build it, and/or fix it. Others will contribute to the process, but for the most part, a developer's life is a negotiation between the brain and the keyboard, and the code.

Management has a different problem. Thinking can beget some good ideas, to be sure, but ideas die stillborn if they're not executed into existence. You think it's a good idea to extend the build system into a full-blown DevOps pipeline? Sure! Now, how do you make that happen? The Operations team isn't just waiting by the phone for good ideas to call up. They have their own concerns and their own deadlines to meet. You need to go to them, pitch the idea, get their buy-in, and find out what obstacles stand in the way. They don't have anybody who can help get the necessary tools in place, and there's no room for them to hire in any new people to own it. It's time to negotiate: Can a developer from your team meet with them? Can you bring in an industry expert to do some training that includes the Ops team? And can your team take time away

(Continued on page 73)

CODE - More Than Just CODE Magazine



The CODE brand is widely-recognized for our ability to use modern technologies to help companies build better software. CODE is comprised of five divisions - CODE Consulting, CODE Staffing, CODE Framework, CODE Training, and CODE Magazine. With expert developers, a repeatable process, and a solid infrastructure, we will exceed your expectations. But don't just take our word for it - ask around the community and check our references. We know you'll be impressed.

Contact us for your free 1-hour consultation.

Helping Companies Build Better Software Since 1993

www.codemag.com
832-717-4445 ext. 9 • info@codemag.com

CODE
An EPS Brand