8

Univerzalni JavaScript za veb aplikacije

Jezik JavaScript je rođen 1995. godine s prvobitnim ciljem da autorima veb aplikacija omogući da izvršavaju kôd direktno u čitaču veba i grade dinamičnije i interaktivnije veb lokacije.

Od tada, JavaScript je znatno proširen i danas je jedan od najpoznatijih i najpopularnijih programskih jezika u svetu. Ako je, u početku, JavaScript bio vrlo jednostavan i ograničen jezik, danas se može smatrati potpunim jezikom opšte namene koji se može upotrebiti i izvan čitača veba za izradu gotovo svake vrste aplikacije. U stvari, JavaScript danas čini „pogon" korisničkih aplikacija, veb servera i mobilnih aplikacija, kao i ugrađenih uređaja kao što su prenosivi uređaji, termostati i leteći dronovi.

Ta mogućnost upotrebe na raznim platformama i uređajima podstiče nov trend među JavaScript programerima, a to je mogućnost višekratne upotrebe istog koda u različitim okruženjima, u okviru istog projekta. Najznačajniji slučaj u vezi s platformom Node.js tiče se prilike za izradu veb aplikacija gde se isti kôd lako deli između servera (pozadinske komponente) i čitača veba (korisničke komponente aplikacije). Cilj višekratne upotrebe istog koda prvobitno su ljudi opisivali izrazom **Izomorfni JavaScript** (engl. *Isomorphic JS*), ali danas je uobičajeniji naziv **Univerzalni JavaScript** (engl. *Universal JS*).

U ovom poglavlju, istražićemo čudesa Univerzalnog JavaScripta, naročito u oblasti razvoja veb aplikacija, i otkriti mnogobrojne alatke i tehnike za deljenje najvećeg dela našeg koda između servera i čitača veba.

Konkretno, saznaćemo kako se isti moduli mogu koristiti i na serverskoj i na klijentskoj strani i kako se koriste alatke kao što su **Webpack** i **Babel** za pakovanje modula za čitač veba. Iskoristićemo biblioteku React i druge poznate module za izradu veb interfejsa, delićemo stanje veb servera s korisničkom komponentom aplikacije, a proučićemo i nekoliko zanimljivih rešenja za univerzalno usmeravanje i univerzalno učitavanje podataka u našim aplikacijama.

Pri kraju ovog poglavlja, trebalo bi da znamo kako se pomoću biblioteke React pišu **jednostranične aplikacije** (engl. *Single-Page Application, SPA-s*) koje najvećim delom koriste već postojeći kôd na našem Node.js serveru, čime se dobijaju aplikacije koje su usklađenije i lakše za razumevanje i održavanje.

Deljenje koda s čitačem veba

Jedan od glavnih razloga popularnosti platforme Node.js jeste činjenica da je napisana na JavaScriptu i koristi V8, što je mašina na kojoj radi i jedan od najpopularnijih čitača veba: Chrome. Možda biste pomislili da je sama ta činjenica dovoljna da deljenje koda između platforme Node.js i čitača veba bude jednostavan posao; međutim, kao što ćemo videti, to nije uvek tačno, osim kada delimo samo kratke, samodovoljne i generičke odlomke koda. Razvijanje koda i za klijentski i za serverski deo aplikacije zahteva ulaganje nezanemarljive količine truda da bismo obezbedili da isti kôd radi kako treba u dva znatno različita okruženja. Na primer, na platformi Node.js nemamo na raspolaganju DOM, niti dugotrajne prikaze, dok u čitaču veba svakako nemamo sistem datoteka, niti mogućniost da pokrećemo nove procese. Osim toga, moramo uzeti u obzir i to da na platformi Node.js možemo bezbedno koristiti mnoge od novih mogućnosti koje uvodi verzija ES2015. Ali to ne možemo raditi i u čitaču veba jer većina čitača veba i dalje koristi verziju ES5, pa će zato izvršavanje ES5 koda na klijentskoj strani ostati najbezbednija opcija još relativno dugo pre nego što čitači veba koji podržavaju ES2015 postanu uobičajeni.

Dakle, cilj većine napora koje ćemo morati da uložimo kada razvijamo za obe platforme biće da te razlike svedemo na minimum. To se može izvesti pomoću apstrakcija i obrazaca koje aplikaciji omogućavaju da bira, dinamički ili u vreme kompajliranja, između koda koji je kompatibilan s čitačem veba i Node.js koda.

Srećom, zbog rastućeg interesa za tu novu fantastičnu mogućnost, mnoge biblioteke i frejmvorkovi u ekosistemu počeli su da podržavaju oba okruženja. Tu evoluciju prati i rastući broj alatki za podršku tom novom načinu rada, koje tokom godina postaju sve rafinisanije i sve savršenije. To znači da ako na platformi Node.js koristimo neki npm paket, postoji velika verovatnoća da će isti paket raditi besprekorno i u čitaču veba. Međutim, to često nije dovoljno jaka garancija da naša aplikacija može da radi bez problema i u čitaču veba i kao Node.js aplikacija. Kao što ćemo videti, kada se razvija kôd za više različitih platformi, uvek je potreban brižljivo osmišljen dizajn.

U ovom odeljku, razmotrićemo najvažnije probleme na koje možemo naići kada pišemo kôd i za Node.js i za čitač veba i predložićemo neke alatke i obrasce koji će nam pomoći da krenemo u taj nov i uzbudljiv izazov.

Deljenje istih modula

Prvi zid na koji naletimo kada želimo da delimo neki kôd između čitača veba i servera jeste neusklađenost između sistema modula na platfrom Node.js i šarenila sistema modula koje koristi čitač veba. Još jedan problem jeste to što u čitaču veba nemamo funkciju `require()`, niti sistem datoteka u kojem bismo razrešavali module. Zbog toga, ako želimo da pišemo velike delove koda koji treba da radi na obe platforme i želimo da nastavimo upotrebu sistema modula CommonJS, potreban nam je dodatan korak—alatka koja nam pomaže da u vreme kompajliranja aplikacije grupišemo sve zavisnosti u jednu celinu i tako apstrahujemo mehanizam `require()` u čitaču veba.

Univerzalna definicija modula

Na platformi Node.js, znamo da su CommonJS moduli standardni mehanizam za deklarisanje zavisnosti između komponenata. Nažalost, situacija unutar *prostora čitača veba* (engl. *browser-spacr*) je znatno rascepkanija:

- Možemo imati okruženje u kojem ne postoji nikakav sistem modula, što znači da su globalni objekti jedini mehanizam za pristupanje drugim modulima.
- Možemo imati okruženje koje podržava neki **AMD** (**Asynchronous Module Definition** – definicija asihronog modula) učitavač modula, kao što je, na primer, RequireJS (`http://requirejs.org`).
- Možemo imati okruženje koje apstrahuje sistem modula CommonJS.

Srećom, postoji obrazac nazvan **univerzalna definicija modula** (engl. *Universal Module Definition*, *UMD*) koji nam može pomoći da naš kôd apstrahujemo od sistema modula koji postoji u okruženju u kojem će taj kôd raditi.

Izrada UMD modula

Pošto UMD još nije sasvim standardizovan, može biti više varijanti koje zavise od potreba određene komponente i sistema modula koji treba podržati. Međutim, postoji jedan oblik koji je verovatno najpopularniji i omogućava da podržimo najuobičajenije sisteme modula, kao što su AMD, CommonJS i globalni objekti u čitaču veba.

Razmotrimo jednostavan primer kako to izgleda. U novom projektu napravićemo modul nazvan `umdModule.js`:

```
(function(root, factory) {                              //[1]
  if(typeof define === 'function' && define.amd) {      //[2]
    define(['mustache'], factory);
  } else if(typeof module === 'object' &&               //[3]
      typeof module.exports === 'object') {
    var mustache = require('mustache');
    module.exports = factory(mustache);
  } else {                                              //[4]
    root.UmdModule = factory(root.Mustache);
  }
}(this, function(mustache) {                            //[5]
  var template = '<h1>Pozdravlja vas <i>{{name}}</i></h1>';
  mustache.parse(template);

  return {
    sayHello:function(toWhom) {
      return mustache.render(template, {name: toWhom});
    }
  };
}));
```

Prethodni primer definiše jednostavan modul koji zavisi od jedne spoljašnje komponente: `mustache` (`http://mustache.github.io`), što je jednostavna mašina za šablone. Konačan proizvod prethodnog UMD modula jeste objekat s jednom

metodom koja se zove `sayHello()` i koja formira `mustache` šablon i vraća ga svom pozivaocu. Cilj UMD-a jeste integrisanje modula s drugim sistemima modula na raspolaganju u okruženju. Evo kako to radi:

1. Ceo kôd je smešten u anonimnu samoizvršljivu funkciju, vrlo sličnu obrascu **otkrivajući modul** koji smo razmatrali u poglavlju 2. Ta funkcija prihvata kao svoj argument objekat `root` koji predstavlja globalni imenski prostor na raspolaganju u sistemu (na primer, u čitaču veba to je objekat `window`). To je potrebno prvenstveno zbog registrovaja zavisnosti u obliku globalne promenljive, kao što ćemo videti u nastavku teksta. Drugi argument je fabrička funkcija `factory()` za modul, koja prihvata kao svoj argument zavisnosti modula (umetanje zavinosti), a vraća instancu modula.
2. Prvo proveravamo da li je AMD na raspolaganju u sistemu. To radimo tako što ispitujemo da li postoji funkcija `define` i njen indikator `amd`. Ako postoji, to znači da u sistemu imamo AMD učitavač modula, pa zato nastavljamo tako što registrujemo naš modul pomoću funkcije `define` i zahtevamo da se zavisnost `mustache` prosledi fabričkoj funkciji `factory()`.
3. Zatim utvrđujemo da li se nalazimo u okruženju nalik Node.js koje podržava sistem modula CommonJS, tako što ispitujemo da li postoje objekti `module` i `module.exports`. Ako postoje, pomoću funkcije `require()` učitavamo sve zavisnosti modula i prosleđujemo ih fabričkoj funkciji `factory()`. Povratnu vrednost te funkcije zatim dodeljujemo objektu `module.exports`.
4. I najzad, ako ne nađemo ni AMD, ni CommonJS, modul dodeljujemo globalnoj promenljivoj, pomoću objekta `root`, što će u okruženju čitača veba obično biti objekat `window`. Osim toga, vidljivo je i to da da se zavisnost, `mustache`, očekuje u obliku objekta u globalnom opsegu vidljivosti.
5. Kao poslednji korak, anonimna omotačka funkcija poziva samu sebe, pri čemu sebi prosleđuje objekat `this` na mestu objekta `root` (u čitaču veba, to će biti objekat `window`), a kao drugi argument prosleđuje fabričku funkciju za naš modul. Vidljivo je da ta funkcija prihvata zavisnosti modula kao svoje argumente.

Vredi istaći i to da da u modulu nismo iskoristili nijednu ES2015 mogućnost. To garantuje da će kôd raditi besprekorno čak i u čitaču veba bez ikakve izmene.

Pogledajmo sada kako se taj UMD modul koristi i na Node.js serveru i u čitaču veba. Prvo ćemo napraviti novu datoteku `testServer.js`:

```
const umdModule = require('./umdModule');
console.log(umdModule.sayHello('server!'));
```

Ako izvršimo ovaj skript, dobićemo sledeći rezultat:

```
<h1>Pozdravlja vas <i>server!</i></h1>
```

Ako ovaj naš sasvim sveže pripremljen modul želimo da upitrebimo i na klijentskoj strani, možemo napraviti stranicu `testBrowser.html` sa sledećim sadržajem:

```
<html>
  <head>
    <script src="node_modules/mustache/mustache.js"></script>
    <script src="umdModule.js"></script>
  </head>
  <body>
    <div id="main"></div>
    <script>
       document.getElementById('main').innerHTML =
         UmdModule.sayHello('čitač veba!');
    </script>
  </body>
</html>
```

Rezultat je veb stranica s velikim naslovom **Pozdravlja vas čitač veba!**. Ovde se dogodilo to da smo naše zavisnosti (`mustache` i naš modul `umdModule`) umetnuli kao standardne skriptove u zaglavlje stranice a zatim smo definisali kratak lokalni skript koji koristi `UmdModule` (na raspolaganju u obliku globalne promenljive u čitaču veba) da bismo generisali malo HTML koda unutar bloka `main`.

Među pratećim primerima koda uz izvorno izdanje ove knjige na veb lokaciji Packt Publishing, naći ćete i druge primere koji prikazuju kako se UMD modul koji smo upravo napravili može upotrebiti i u kombinaciji s nekim AMD učitavačem i sistemom modula CommonJS.

Napomene u vezi s obrascem UMD

Obrazac UMD je efikasna i jednostavna tehnika za izradu modula koji je kompatibilan s najpopularnijim sistemima modula koji zasad postoje. Međutim, videli smo da to zahteva mnogo šablonskog koda koji se ponavlja, što može biti teško za testiranje u svakom okruženju i neizbežno je podložno greškama. To znači da ručno pisanje UMD šablona može imati smisla ako se tako umotava samo jedan modul koji je već ranije bio razvijen i testiran. To nije preporučljiva praksa kada pišemo nov modul od početka; pošto je to neizvodljivo i nepraktično, u takvim situaicjama je bolje da taj posao prepustite altkama koje automatizuju ceo postupak. Jedna od tih alatki je Webpack, koju ćemo koristiti u ovom poglavlju.

Treba napomenuti i to da AMD, CommonJS i globalni objekti u čitaču veba nisu jedini sistemi modula koji postoje. Obrazac koji smo predstavili pokriva većinu slučajeva upotrebe, ali zahteva izmene da bi podržao drugi sistem modula. Na primer, ES2015 specifikacija modula je nešto što ćemo razmatrati u sledećem odeljku ovog poglavlja budući da pruža više prednosti u poređenju s drugim rešenjima a i već je sastavni deo novog standarda ECMAScript (uprkos tome što u vreme pisanja ove knjige nije standardno podržana na platformi Node.js).

Opsežnu listu formalizovanih UMD obrazaca možete naći na `https://github.com/umdjs/umd`.

ES2015 moduli

Jednu od mogućnosti koje je uvela specifikacija ES2015 jeste **ugrađeni sistem modula** (engl. *built-in module system*). Ovo je prvo mesto u ovoj knjizi gde se on pominje zato što, nažalost, u vreme pisanja ove knjige, tekuća verzija Node.js još ne podržava ES2015 module.

Ovu mogućnost nećemo detaljno opisivati ali je važno znati da ona postoji zato što će ona u narednim godinama najverovatnije postati preporučena sintaksa za module. Osim što su standardizovani, ES2015 moduli uvode lepšu sintaksu i pružaju više drugih prednosti u poređenju s drugim sistemima modula koje smo razmatrali.

Cilj ES2015 modula bio je da se preuzme ono najbolje od CommonJS i AMD modula:

- Slično CommonJS-u, ova specifikacija pruža kompaktnu sintaksu, preporučuje izvoženje samo po jedne funkcionalnosti i podržava uzajamne zavisnosti.
- Sličnu AMD-u, nudi direktnu podršku za asinhrono učitavanje i podesiv način učitavanja modula.

Osim toga, zahvaljujući deklarativnoj sintaksi, moguća je upotreba statičkih analizatora za poslove kao što su statičke provere i optimizacije. Na primer, možete analizirati stablo zavisnosti skripta i umetnuti ih u datoteku za čitač veba iz koje su uklonjene sve neupotrebljene funkcije uvezenih modula, čime se na klijentskoj strani dobija kompaktnija datoteka i smanjuje vreme učitavanja.

Da biste saznali više o sintaksi ES2015 modula, pogledajte u ES2015 specifikaciji: `http://www.ecma-international.org/ec ma-262/6.0/#sec-scripts-and-modules`.

Danas možete koristiti i novu Node.js sintaksu za module, pomoću nekog transpajlera kao što je Babel. Mnogi autori aplikacija ga zapravo preporučuju kada predstavljaju rešenja za izradu Universal JavaScript aplikacija. Voditi računa i o budućnosti je uglavnom dobra ideja, posebno zato što je ova mogućnost već standardizovana i vremenom će postati sastavni deo Node.js jezgra. Jednostavnosti radi, u celom ovom poglavlju nastavićemo da koristimo sintaksu CommonJS.

Uvod u Webpack

Kada pišemo Node.js aplikaciju, poslednje što bismo poželeli jeste da ručno dodajemo podršku za određeni sistem modula koji je različit od onog koji nam platforma standardno nudi. Savršen slučaj bio bi da nastavimo pisanje naših modula na način na koji uvek radimo, pomoću `require()` i `module.exports`, a da zatim

pomoću određene alatke transformišemo naš kôd tako da ga može izvršavati i čitač veba. Srećom, taj problem su već rešili mnogi projekti, među kojima je Webpack (`https://webpack.github.io`) jedan od najpopularnijih i naširoko prihvaćen.

Webpack omogućava da pišemo module koristeći Node.js konvencije za module i da zatim, zahvaljujući koraku kompajliranja, formiramo *svežanj* (engl. *bundle*), što je JavaScript datoteka koja sadrži sve zavisnosti koje su našim modulima potrebne da bi radili u čitaču veba (uključujući tu i apstrakciju funkcije `require()`). Taj svežanj se zatim može umetnuti u veb stranicu i izvršavati unutar čitača veba. Webpack rekurzivno pregleda naš izvorni kôd i traži reference na funkciju `require()`, a zatim razrešava i uključuje referencirane module u svežanj.

Webpack nije jedina alatka koju imamo za izradu svežnjeva Node.js modula namenjenih čitaču veba. Druge popularne alternative su `Browseriify` (`http://browserify.org`), `RollupJs` (`http://rollupjs.org`) i `Webmake` (`https://npmjs.org/package/webmake`). Osim toga, `require.js` omogućava izradu modula i za klijentsku stranu i za Node.js ali koristi AMD umesto CommonJSa (`http://requirejs.org/docs/node.html`).

Čarolija Webpacka

Da bismo odmah videli kako deluje ta čarolija, pogledajmo kako izgleda modul `umdModule`, koji smo napravili u prethodnom odeljku, kada upotrebimo Webpack. Prvo moramo instalirati sam Webpack; to se radi pomoću sledeće jednostavne komande:

```
npm install webpack -g
```

Opcija `-g` nalaže `npmu` da instalira Webpack globalno, što će nam omogućiti da mu pristupamo pomoću jednostavne komande na konzoli, kao što ćemo videti u nastavku teksta.

Dalje, započećemo nov projekat i pokušati da napišemo modul ekvivalentan modulu `umdModule` koji sno ranije napisali. Evo kako to izgleda ako treba da ga implementiramo na platformi Node.js (datoteka `sayHello.js`):

```
var mustache = require('mustache');
var template = '<h1>Pozdravlja vas <i>{{name}}</i></h1>';
mustache.parse(template);
module.exports.sayHello = function(toWhom) {
  return mustache.render(template, {name: toWhom});
};
```

Ovo je svakako jednostavnije od primene obrasca UMD, zar ne? Sada ćemo napisati datoteku `main.js`, odnosno ulaznu tačku koda u naš čitač veba:

```
window.addEventListener('load', function(){
  var sayHello = require('./sayHello').sayHello;
  var hello = sayHello('čitač veba!');
  var body = document.getElementsByTagName("body")[0];
  body.innerHTML = hello;
});
```

U prethodnom kodu, zahtevamo modul `sayHello` na potpuno istovetan način kao što bismo to uradili za Node.js, pa zato nema više petljanja sa upravljanjem zavisnostima, niti s konfigurisanjem putanja; jednostavan `require()` je sve što treba.

Dalje, moramo instaiirati u projekat i paket `mustache`:

```
npm install mustache
```

A sada dolazi čarolija. Na komandnoj liniji, zadajte sledeću komandu:

```
webpack main.js bundle.js
```

Prethodna komanda će kompajlirati modul `main` i zajedno sa svim zavisnostima koje su mu potrebne napraviti jednu datoteku čije je ime `bundle.js`, spremnu za upotrebu u čitaču veba!

Da bismo na brzinu testirali tu tvrdnju, napisaćemo HTML stranicu `magic.html` koja sadrži sledeći kôd:

```
<html>
  <head>
    <title>Webpack čarolija</title>
    <script src="bundle.js"></script>
  </head>
    <body>
    </body>
</html>
```

To je sve što treba da bi naš kôd radio u čitaču veba. Otvorite tu stranicu i uverite se i sami. Bum!

Tokom razvoja koda, ne želimo da ručno pokrećemo Webpack pri svakoj izmeni koju načinimo u izvornom kodu. Umesto toga korisniji bi nam bio neki automatski mehanizam za ponovno formiranje svežnja kada se naš izvorni kôd izmeni. Da bismo to postigli, kada zadajemo komandu Webpack, imamo opciju `--watch`. Ta opcija čini da Webpack radi neprekidno i sam ponovo kompajlira svežanj kad god se promeni jedna od datoteka s izvornom kodom.

Prednosti upotrebe Webpacka

Čarolija Webpacka se ne zaustavlja ovde. Sledi (nepotpuna) lista mogućnosti koje deljenje koda s čitačem čine jednostavnim poslom:

- Webpack automatski obezbeđuje verzije mnogih osnovnih Node.js modula koje su kompatibilne s čitačem veba. To znači da u čitaču veba možemo koristiti i module kao što su `http`, `assert` ili `events`, kao i mnoge druge!

Modul `fs` je jedan od onih koji nisu podržani.

- Ako imamo modul koji nije kompatibilan s čitačem veba, možemo ga isključiti iz svežnja ili zameniti praznim objektom ili drugim modulom koji pruža alternativnu implementaciju kompatibilnu s čitačem veba. To je ključna mogućnost koju ćemo imati priliku da iskoristimo u prmeru koji ćemo videti u nastavku teksta.
- Webpack može da generiše svežnjeve za različite module.
- Webpack omogućava dopunsku obradu izvornih datoteka pomoću **učitavača** i **priključnih dodataka** iz trećih izvora. Postoje učitavači i priključni dodaci za gotovo sve što bi moglo da nekome zatreba, od CoffeeScript, TypeScript ili ES2015 kompilacije, do podrške za učitavanje AMD, Bower (`http://bower.io`) i Component (`http://component.github.io`) paketa pomoću funkcije `require()`, od minifikacije do kompajliranja i umetanja u svežnjeve drugih elemenata kao što su šabloni i liste stilova.
- Webpack se lako može pozivati iz upravljača poslovima kao što su Gulp (`https://npmjs.com/package/gulp-webpack`) i Grunt (`https://npmjs. org/package/grunt-webpack`).
- Webpack omogućava preprocesiranje i upravljanje svim resursima jednog projekta, ne samo JavaScript datotekama, nego i listama stilova, slikama, fontovima i šablonima.
- Webpack se može konfigurisati i tako da podeli stablo zavisnosti i organizuje ga u drugačije delove koji se mogu učitavati na zahtev kad god zatrebaju u čitaču veba.

Moć i fleksibilnost Webpacka su toliko privlačni da su mnogi programeri počeli da ga koriste čak i za upravljanje kodom koji je namenjen isključivo za klijentsku stranu. To je omogućila činjenica da mnoge biblioteke za klijentsku stranu počinju da standardno podržavaju CommonJS i npm, što otvara nova i zanimljiva scenarija. Na primer, jQuery možemo instalirati na sledeći način:

```
npm install jquery
```

Nakon toga, možemo ga učitati u naš kôd pomoću sledećeg reda jednostavnog koda:

```
const $ = require('jquery');
```

Iznenadili biste se koliko biblioteka za klijentsku stranu već podržava CommonJS i Webpack.

Primena standarda ES2015 s Webpackom

Kao što smo već pomenuli u prethodnom odeljku, jedan od glavnih prednosti Webpacka jeste mogućnost upotrebe učitavača i priključnih dodataka za transformisanje izvornog koda pre kompajliranja.

U ovoj knjizi dosad smo koristili mnoge zgodne nove mogućnosti koje pruža standard ES2015, a voleli bismo da nastavimo njihovu upotrebu čak i kada radimo na Universal JavaScript aplikaciji. U ovom odeljku, videćemo kako možemo iskoristiti mogućnost upotrebe učitavača u Webpacku da bismo drugačije napisa-

li prethodni primer pomoću ES2015 sintakse u izvornom kodu naših modula. Uz odgovarajuće konfigurisanje, Webpack će se pobrinuti za transpajliranje rezultujućeg koda za čitač veba u ES5 verziju, da bi garantovao maksimalnu kompatibilnost sa svim čitačima veba koji su trenutno na raspolaganju.

Prvo i pre svega, premestićemo naše module u nov direktorijum `src`. To će nam olakšati organizovanje našeg koda i razdvajanje transpajlovane datoteke od prvobitne verzije izvornog koda. To razdvajanje će nam takođe omogućiti da lakše konfigurišemo Webpack kako treba i da pojednostavimo način na koji pozivamo Webpack sa komandne linije.

Sada smo spremni za pisanje naših modula. ES2015 verzija naše datoteke `src/sayHello.js` izgleda ovako:

```
const mustache = require('mustache');
const template = '<h1>Pozdravlja vas <i>{{name}}</i></h1>';
mustache.parse(template);
module.exports.sayHello = toWhom => {
  return mustache.render(template, {name: toWhom});
};
```

Obratite pažnju na to da koristimo `const`, `let` i sintaksu streličastih funkcija.

Sada možemo da prepravimo i našu datoteku `src/main.js` u ES2015 verziju. Njen sadržaj bi bio sledeći:

```
window.addEventListener('load', () => {
  const sayHello = require('./sayHello').sayHello;
  const hello = sayHello('čitač veba!');
  const body = document.getElementsByTagName("body")[0];
  body.innerHTML = hello;
});
```

Sada možeo definisati i datoteku `webpack.config.js`:

```
const path = require('path');

module.exports = {
  entry: path.join(   dirname, "src", "main.js"),
  output: {
    path: path.join(   dirname, "dist"),
    filename: "bundle.js"
  },
  module: {
    loaders: [
      {
        test: path.join(   dirname, "src"),
        loader: 'babel-loader',
        query: {
          presets: ['es2015']
        }
      }
    ]
  }
};
```

Ova datoteka je modul koji izvozi konfiguracijski objekat koji Webpack učitava kad ga pozovemo sa komandne linije bez argumenata.

U konfiguracijskom objektu, kao ulaznu tačku definišemo našu datoteku `src/main.js`, a kao odredišni svežanj zadajemo našu datoteke `dist/bundle.js`.

Pošto ovaj deo objašnjava sam sebe prilično dobro, razmotrićemo niz učitavača `loaders`. Taj opcioni niz omogućava da zadamo skup učitavača koji menjaju sadržaj naših datoteka izvornog koda kada Webpack konstruiše datoteku svežnja. Ideja se sastoji u tome da svaki učitavač predstavlja određenu transformaciju (u ovom slučaju, ES2015 u ES5 verziju, pomoću alatke `babel-loader`) koja se obavlja samo ako tekuća izvorna datoteka ispunjava uslov definisan izrazom `test` koji je zadat učitavaču. U ovom primeru, nalažemo Webpacku da pomoću alatke `babel-loader` obradi sve datoteke koje se nalaze u našem direktorijumu `src` i da pri tome koristi uz Babel preset opciju `es2015`.

Sada smo gotovo spremni; jedini nedostajući korak pre nego što pokrenemo Webpack jeste da instaliramo Babel i ES2015 preset pomoću sledeće komande:

```
npm install babel-core babel-loader babel-preset-es2015
```

Da biste generisali svežanj, sada treba samo da zadate:

```
webpack
```

Ne zaboravite da u svojoj datoteci `magic.html` referencirate novu datoteku `dist/bundle.js`. Trebalo bi da možete da je otvorite u čitaču veba i uverite se da sve i dalje ispravno radi.

Ako ste radoznali, možete pogledati sadržaj generisane datoteke svežnja. Otkrićete da su ES2015 mogućnosti koje smo koristili u izvornim datotekama konvertovane u ekvivalentan kôd ispravan u ES5 verziji, koji svaki čitač na tržištu izvršava bez problema.

Osnove razvoja aplikacija za ciljnu platformu različitu od izvorne

Kada razvijamo aplikacije za ciljnu platformu različitu od izvorne, najuobičajeniji problem na koji nailazimo jeste kako da zadržimo zajedničke delove jedne komponente koji su nezavisno od platforme, ali i da istovremeno obezbedimo implementacije za detalje koji su specifični za određenu platformu. Sada ćemo razmotriti neke principe i obrasce koje možemo iskoristiti kada naiđemo na taj problem.

Grananje koda u vreme izvršavanja

Najjednostavniaj i najintuitivnija tehnika za obezbeđivanje različitih implementacija na osnovu ciljne platforme jeste dinamičko grananje našeg koda, u vreme izvršavanja. To zahteva postojanje određenog mehanizma koji u vreme izvršavanja koda prepoznaje ciljnu platformu a zatim pomoću naredbe `if…else` dinamički uključuje odgovarajuću implementaciju. Neka generička rešanja rade tako što ispituju postojanje globalnih promenljivih koje postoje samo na platformi Node.js