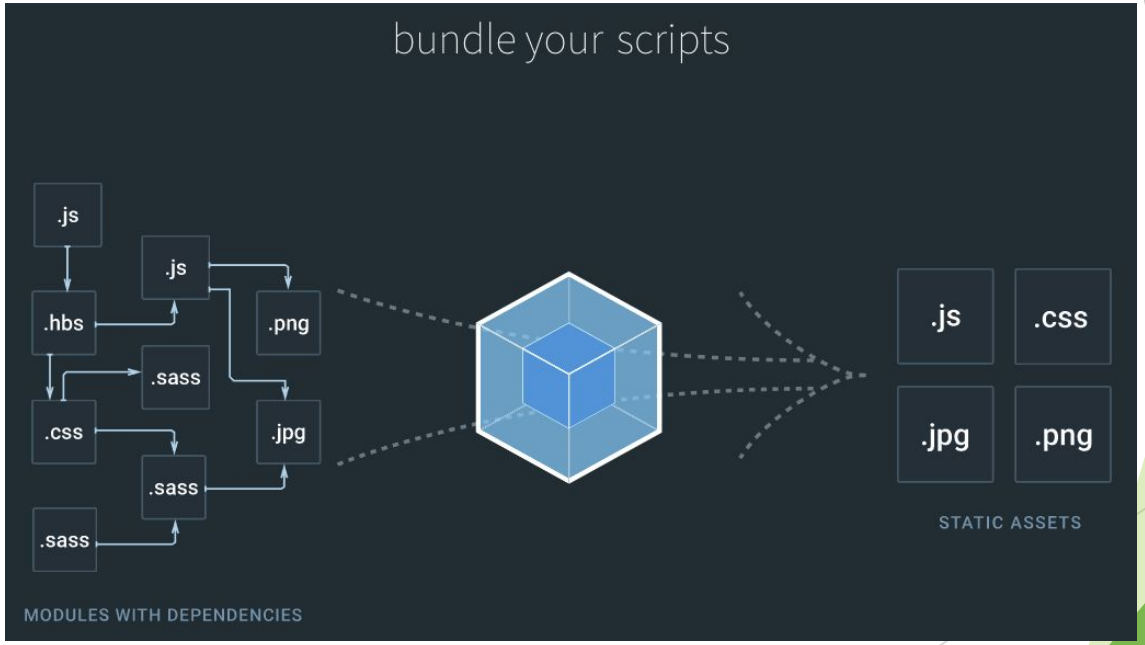
Webpack


bundle your scripts
.js
.js
.hbs
.png
.sass
.css
.jpg
.sass
.sass
MODULES WITH DEPENDENCIES
.js
.css
.jpg
.png
STATIC ASSETS

Webpack

webpack is a ***module bundler*** for modern JavaScript applications. When webpack processes your application, it recursively builds a *dependency graph* that includes every module your application needs, then packages all of those modules into a small number of *bundles* - often only one - to be loaded by the browser.

Final Result

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); //installed via npm
const webpack = require('webpack'); //to access built-in plugins
const path = require('path');

const config = {
  entry: './path/to/my/entry/file.js',
  output: {
    filename: 'my-first-webpack.bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        use: 'babel-loader'
      }
    ]
  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin(),
    new HtmlWebpackPlugin({template: './src/index.html'})
  ]
};

module.exports = config;
```

Core Concepts

- **Entry:** webpack creates a graph of all of your application's dependencies. The starting point of this graph is known as an *entry point*. The *entry point* tells webpack *where to start* and follows the graph of dependencies to know *what to bundle*. You can think of your application's *entry point* as the contextual root or the first file to kick off your app.

```
module.exports = {
  entry: './path/to/my/entry/file.js'
};
```

- **Output:** Once you've bundled all of your assets together, you still need to tell webpack where to bundle your application. The webpack output property tells webpack how to treat bundled code.

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

module.exports = {
  entry: './path/to/my/entry/file.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-first-webpack.bundle.js'
  }
};
```

Core Concepts

- **Loaders:** The goal is to have all of the assets in your project be webpack's concern and not the browser's (though, to be clear, this doesn't mean that they all have to be bundled together). webpack treats every file (.css, .html, .scss, .jpg, etc.) as a module. However, webpack itself only understands JavaScript. Loaders in webpack *transform these files into modules* as they are added to your dependency graph.

```
module: {
  rules: [
    { test: /\.txt$/, use: 'raw-loader' }
  ]
}
```

- **Plugins:** While Loaders only execute transforms on a per-file basis, plugins are most commonly used to perform actions and custom functionality on "compilations" or "chunks" of your bundled modules (and so much more!). The webpack Plugin system is extremely powerful and customizable.

```
plugins: [
  new webpack.optimize.UglifyJsPlugin(),
  new HtmlWebpackPlugin({template: './src/index.html'})
]
```

Entry

There are multiple ways to define the entry property in webpack configuration.

- **Single Entry (Shorthand) Syntax**

  Usage: entry: string|Array<string>

```
const config = {
  entry: './path/to/my/entry/file.js'
};

module.exports = config;
```

The single entry syntax for the entry property is a shorthand for:

```
const config = {
  entry: {
    main: './path/to/my/entry/file.js'
  }
};
```

Passing an array of file paths to the entry *property creates what is known as a* ***"multi-main entry"****. This is useful when you would like to inject multiple dependent files together and graph their dependencies into one "chunk".*

Entry

- **Object Syntax**

  Usage: entry: {[entryChunkName: string]: string|Array<string>}

```
const config = {
  entry: {
    app: './src/app.js',
    vendors: './src/vendors.js'
  }
};
```

  This is the most scalable way of defining entry/entries in your application.

  *"**Scalable webpack configurations**" are ones that can be reused and combined with other partial configurations. This is a popular technique used to separate concerns by environment, build target and runtime. They are then merged using specialized tools like webpack-merge.*

Scenarios

Below is a list of entry configurations and their real-world use cases:

- **Separate App and Vendor Entries**

  Usage: entry: {[entryChunkName: string]: string|Array<string>}

```
const config = {
  entry: {
    app: './src/app.js',
    vendors: './src/vendors.js'
  }
};
```

  - **What does this do?** At face value this tells webpack to create dependency graphs starting at both app.js and vendors.js. These graphs are completely separate and independent of each other (there will be a webpack bootstrap in each bundle). This is commonly seen with single page applications which have only one entry point (excluding vendors).
  - **Why?** This setup allows you to leverage CommonsChunkPlugin and extract any vendor references from your app bundle into your vendor bundle, replacing them with __webpack_require__() calls. If there is no vendor code in your application bundle, then you can achieve a common pattern in webpack known as long-term vendor-caching.

Scenarios

- **Multi Page Application**

  Usage: entry: {[entryChunkName: string]: string|Array<string>}

```
const config = {
  entry: {
    pageOne: './src/pageOne/index.js',
    pageTwo: './src/pageTwo/index.js',
    pageThree: './src/pageThree/index.js'
  }
};
```

  - **What does this do?** We are telling webpack that we would like 3 separate dependency graphs (like the above example).
  - **Why?** In a multi-page application, the server is going to fetch a new HTML document for you. The page reloads this new document and assets are redownloaded. However, this gives us the unique opportunity to do multiple things:
    - Use CommonsChunkPlugin to create bundles of shared application code between each page. Multi-page applications that reuse a lot of code/modules between entry points can greatly benefit from these techniques, as the amount of entry points increase.

Output

Configuring the output configuration options tell webpack how to write the compiled files to disk. Note that, while there can be multiple entry points, only one output configuration is specified.

- **Usage**

  The minimum requirements for the output property in your webpack config is to set its value to an object including the following two things:

  - A filename to use for the output file(s).
  - An absolute path to your preferred output directory.

```
const config = {
  output: {
    filename: 'bundle.js',
    path: '/home/proj/public/assets'
  }
};

module.exports = config;
```

Output

- **Multiple Entry Points**

  If your configuration creates more than a single "chunk" (as with multiple entry points or when using plugins like CommonsChunkPlugin), you should use substitutions to ensure that each file has a unique name.

```
{
  entry: {
    app: './src/app.js',
    search: './src/search.js'
  },
  output: {
    filename: '[name].js',
    path: __dirname + '/dist'
  }
}

// writes to disk: ./dist/app.js, ./dist/search.js
```

Loaders

Loaders are transformations that are applied on the source code of a module. They allow you to pre-process files as you import or “load” them. Thus, loaders are kind of like “tasks” in other build tools, and provide a powerful way to handle front-end build steps. Loaders can transform files from a different language (like TypeScript) to JavaScript, or inline images as data URLs. Loaders even allow you to do things like import CSS files directly from your JavaScript modules!

- **Example**

  For example, you can use loaders to tell webpack to load a CSS file or to convert TypeScript to JavaScript. To do this, you would start by installing the loaders you need:

```
npm install --save-dev css-loader
npm install --save-dev ts-loader
```

```
module.exports = {
  module: {
    rules: [
      { test: /\.css$/, use: 'css-loader' },
      { test: /\.ts$/, use: 'ts-loader' }
    ]
  }
};
```

Loaders

- **Using Loaders**

  There are three ways to use loaders in your application:

  - Configuration (recommended): Specify them in your **webpack.config.js** file.
  - Inline: Specify them explicitly in each import statement.

    It's possible to specify loaders in an import statement, or any equivalent "importing" method. Separate loaders from the resource with !. Each part is resolved relative to the current directory.

    ```
    import Styles from 'style-loader!css-loader?modules!./styles.css';
    ```

    Options can be passed with a query parameter, e.g. ?key=value&foo=bar, or a JSON object, e.g. ?{"key":"value","foo":"bar"}.
  - CLI: Specify them within a shell command.

    You can also use loaders through the CLI:

    ```
    webpack --module-bind jade-loader --module-bind 'css=style-loader!css-loader'
    ```

    This uses the jade-loader for .jade files, and the style-loader and css-loader for .css files.

```
module: {
  rules: [
    {
      test: /\.css$/,
      use: [
        { loader: 'style-loader' },
        {
          loader: 'css-loader',
          options: {
            modules: true
          }
        }
      ]
    }
  ]
}
```

Loaders

- **Loader Features**
  - Loaders can be chained. They are applied in a pipeline to the resource. A chain of loaders are compiled chronologically. The first loader in a chain of loaders returns a value to the next. At the end loader, webpack expects JavaScript to be returned.
  - Loaders can be synchronous or asynchronous.
  - Loaders run in Node.js and can do everything that's possible there.
  - Loaders accept query parameters. This can be used to pass configuration to the loader.
  - Loaders can also be configured with an options object.
  - Normal modules can export a loader in addition to the normal main via package.json with the loader field.
  - Plugins can give loaders more features.
  - Loaders can emit additional arbitrary files.

Loaders

- **Resolving Loaders**

  Loaders follow the standard module resolution. In most cases it will be loaders from the module path(think npm install, node_modules).

  A loader module is expected to export a function and be written in Node.js compatible JavaScript. They are most commonly managed with npm, but you can also have custom loaders as files within your application. By convention, loaders are usually named xxx-loader (e.g. json-loader).

Plugins

Plugins are the backbone of webpack. webpack itself is built on the same plugin system that you use in your webpack configuration!

They also serve the purpose of doing anything else that a loader cannot do.

- **Anatomy**

  A webpack plugin is a JavaScript object that has an apply property. The apply property is called by the webpack compiler, giving access to the entire compilation lifecycle.

```
function ConsoleLogOnBuildWebpackPlugin() {

};

ConsoleLogOnBuildWebpackPlugin.prototype.apply = function(compiler) {
  compiler.plugin('run', function(compiler, callback) {
    console.log("The webpack build process is starting!!!");

    callback();
  });
};
```

Plugins

- **Usage**

  Since plugins can take arguments/options, you must pass a new instance to the plugins property in your webpack configuration.

  Depending on how you are using webpack, there are multiple ways to use plugins.

Plugins

- Configuration

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); //installed via npm
const webpack = require('webpack'); //to access built-in plugins
const path = require('path');

const config = {
  entry: './path/to/my/entry/file.js',
  output: {
    filename: 'my-first-webpack.bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        use: 'babel-loader'
      }
    ]
  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin(),
    new HtmlWebpackPlugin({template: './src/index.html'})
  ]
};

module.exports = config;
```

Thank you