# Front-end Tricky Questions

1. **package.json Vs package-lock.json**

   `npm i <package-name> — save`

   , it will install the exact latest version of that package in your project and save the dependency in package.json with a carat (^) sign. Like, if the current version of a package is 5.2.3 then the installed version will be 5.2.3 and the saved dependency will be ^5.2.3. Carat (^) means it will support any higher version with major version 5 like 5.3.1 and so on. Here, package-lock.json is created for locking the dependency with the installed version.

   To avoid differences in installed dependencies on different environments and to generate the same results on every environment we should use the package-lock.json file to install dependencies.

   Ideally, this file should be on your source control with the package.json file so when you or any other user will clone the project and run the command “npm i”, it will install the exact same version saved in package-lock.json file and you will able to generate the same results as you developed with that particular package.

   During deployment, when you again run “npm i” with the same package.json file without the package-lock.json, the installed package might have a higher version now from what you had intended.

2. **Difference between tilde (~) and caret (^) in package.json?**

   ~version “Approximately equivalent to version”, will update you to all future patch versions, without incrementing the minor version. ~1.2.3 will use releases from 1.2.3 to <1.3.0.

   ^version “Compatible with version”, will update you to all future minor/patch versions, without incrementing the major version. ^2.3.4 will use releases from 2.3.4 to <3.0.0.

3. **Install a previous exact version of a NPM package?**

   If you have to install an older version of a package, just specify it

   `npm install <package>@<version>`

   For example: `npm install express@3.0.0`

   You can also add the --save flag to that command to add it to your package.json dependencies, or --save --save-exact flags if you want that exact version specified in your package.json dependencies.

4. **Dependencies vs devDependencies in npm package.json file?**

Dependencies are used for direct usage in your codebase, things that usually end up in the production code, or chunks of code

devDependencies are used for the build process, tools that help you manage how the end code will end up, third party test modules, (ex. webpack stuff, unit tests, CoffeeScript to JavaScript transpilation, minification etc.)

If you are going to develop a package, you download it (e.g. via git clone), go to its root which contains package.json, and run:

```
npm install
```

Since you have the actual source, it is clear that you want to develop it, so by default,
both dependencies (since you must, of course, run to develop) and devDependency dependencies are also installed.

- npm install will install both "dependencies" and "devDependencies"
- npm install --production will only install "dependencies"
- npm install --dev will only install "devDependencies"

5. **Do you put Babel and Webpack in devDependencies or Dependencies?**

   The babel and webpack packages will go into the devDependencies section because these packages are used in when transpiling and bundle-ing your code into vanilla javascript in the bundle.js & etc file(s).

   In production you will run your code off the bundle.js build/generated code will not require these dependencies anymore.

6. **npm Install vs npm ci**

| Npm install | Npm ci |
|---|---|
| npm install reads package.json to create a list of dependencies and uses package-lock.json to inform which versions of these dependencies to install. If a dependency is not in package-lock.json it will be added by npm install. | **npm ci (named after Continuous Integration)** installs dependencies directly from package-lock.json and uses package.json only to validate that there are no mismatched versions. If any dependencies are missing or have incompatible versions, it will throw an error. |
| Use npm install to add new dependencies, and to update dependencies on a project. Usually, you would use it during development after pulling changes that update the list of | Use npm ci if you need a deterministic, repeatable build. For example during continuous integration, automated jobs, etc. and when installing dependencies for the first time, instead of npm install. |

| | |
|---|---|
| dependencies but it may be a good idea to use npm ci in this case. | |
| Requires package.json | Requires package-lock.json |
| If a node_modules is already present, it will install any missing dependencies in node_modules | Throws an error if dependencies from package-lock.json file don't match package.json.<br>If a node_modules is already present, it will be automatically removed before npm ci begins its install. |
| may write to package.json or package-lock.json.<br>• When used with an argument (npm i packagename) it may write to package.json to add or update the dependency.<br>• when used without arguments, (npm i) it may write to package-lock.json to lock down the version of some dependencies if they are not already in this file | never writes to package.json or package-lock.json |

**7. Package Managers vs Module Loader/Bundling vs Task runner**

| Package Managers | Module Loader/Bundling | Task runner |
|---|---|---|
| Package managers simplify installing and updating project dependencies, which are libraries such as: jQuery, Bootstrap, etc - everything that is used on your site and isn't written by you. | Most projects of any scale will have their code split between several files. You can just include each file with an individual <script> tag, however, <script> establishes a new HTTP connection, and for small files – which is a goal of modularity – the time to set up the connection can take significantly longer than transferring the data. While the scripts are downloading, no content can be changed on the page.<br>The problem of download time can largely be solved by concatenating a group of simple modules into a single file and minifying it. | Task runners and build tools are primarily command-line tools. Why we need to use them: In one word: automation. The less work you have to do when performing repetitive tasks like minification, compilation, unit testing, linting which previously cost us a lot of times to do with command line or even manually. |
| Examples are NPM, Bower, Yarn | Examples are RequireJS, Browserify, Webpack and SystemJS<br><br>Webpack bundles all of your static assets, including JavaScript, images, CSS, and more, into a single file. | Examples are Grunt, Gulp |

8. **Use of Babel (JavaScript compiler)**

   Lack of browser support for some of the newer features of JavaScript (ES6) has been a major challenge for many front-end developers who want to use the latest and greatest to write better code and ship features faster.

   A transpiler (AKA source-to-source compiler) is a type of compiler that takes the source code of a program in one language and produces the equivalent source code in another language. This means I can write ES6 JavaScript and have it converted to ES5 so that it can run on all modern browsers. I can also choose to write TypeScript, CoffeeScript, ClojureScript or Dart code, taking advantage of features that make it faster and scalable to build large web apps.

   Babel is a commonly used transpiler for converting ES6+ code into a backwards compatible version of JavaScript supported by current and older browsers. It is also used to accomplish specific tasks such as converting JSX in React.js code to JavaScript via plugins.

   **Advantages of using BabelJS**

   In this section, we will learn about the different advantages associated with the use of BabelJS −

   - BabelJS provides backward compatibility to all the newly added features to JavaScript and can be used in any browsers.
   - BabelJS has the ability to transpile to take the next upcoming version of JavaScript - ES6, ES7, ESNext, etc.
   - BabelJS can be used along with gulp, webpack, flow, react, typescript, etc. making it very powerful and can be used with big project making developer's life easy.
   - BabelJS also works along with react JSX syntax and can be compiled in JSX form.
   - BabelJS has support for plugins, polyfills, babel-cli that makes it easy to work with big projects.

   **Disadvantages of using BabelJS**

   In this section, we will learn about the different disadvantages of using BabelJS −

   - BabelJS code changes the syntax while transpiling which makes the code difficult to understand when released on production.
   - The code transpiled is more in size when compared to the original code.
   - Not all ES6/7/8 or the upcoming new features can be transpiled and we have to use polyfill so that it works on older browsers.

   Here is the official site of babeljs https://babeljs.io/

   http://nicholasjohnson.com/blog/what-is-babel/

9. **gitignore file - ignoring files in Git**

https://git-scm.com/docs/gitignore
https://www.atlassian.com/git/tutorials/saving-changes/gitignore

**10. .editorconfig**

EditorConfig helps maintain consistent coding styles for multiple developers working on the same project across various editors and IDEs. The EditorConfig project consists of a file format for defining coding styles and a collection of text editor plugins that enable editors to read the file format and adhere to defined styles. EditorConfig files are easily readable and they work nicely with version control systems.

https://editorconfig-specification.readthedocs.io/

**11. Tslint**

TSLint is an extensible static analysis tool that checks TypeScript code for readability, maintainability, and functionality errors. It is widely supported across modern editors & build systems and can be customized with your own lint rules, configurations, and formatters.

You can use tslint npm package in your project for better coding structure and quality.

```
npm i tslint
```

This tslint package will be installed as devDependencies in package.json.

12. **How can you increase page performance?**
    a. Reusable code structure
    b. Write less code as much as possible
    c. Sprites, compressed images, smaller images.
    d. Incorporate JavaScript at the bottom of the page
    e. Minify CSS, JavaScript, HTML
    f. Caching static contents and images

**13. What is Content Security Policy?**

Content Security Policy (CSP) is an HTTP header that allows site operators fine-grained control over where resources on their site can be loaded from. The use of this header is the best method to prevent cross-site scripting (XSS) vulnerabilities. Due to the difficulty in retrofitting CSP into existing websites, CSP is mandatory for all new websites and is strongly recommended for all existing high-risk sites.

The primary benefit of CSP comes from disabling the use of unsafe inline JavaScript. Inline JavaScript – either reflected or stored – means that improperly escaped user-inputs can generate code that is interpreted by the web browser as JavaScript. By using CSP to disable inline JavaScript, you can effectively eliminate almost all XSS attacks against your site.

## 14. What is Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is an attack that occurs when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user.

The page provided by the server when someone requests it is unaltered. Instead, an XSS attack exploits a weakness in a page that include a variable submitted in a request to show up in raw form in the response. The page is only reflecting back what was submitted in that request.

https://www.acunetix.com/websitesecurity/cross-site-scripting/

## 15. Object oriented languages vs Object-based languages

**Object Based Languages**

- Object based languages supports the usage of object and encapsulation.
- They does not support inheritance or, polymorphism or, both.
- Object based languages does not supports built-in objects.
- Javascript, VB are the examples of object bases languages.

**Object Oriented Languages**

- Object Oriented Languages supports all the features of Oops including inheritance and polymorphism.
- They support built-in objects.
- C#, Java, VB. Net are the examples of object oriented languages.

## 16. RESTful and RESTless web service

**1. Protocol**

- RESTful services use REST architectural style. It uses one and only one protocol – HTTP.
- RESTless services use SOAP protocol.

**2. Business logic / Functionality**

- RESTful services use URL to expose business logic,
- RESTless services use the service interface to expose business logic.

**3. Security**

- RESTful inherits security from the underlying transport protocols,
- RESTless defines its own security layer, thus it is considered as more secure.

**4. Data format**

- RESTful supports various data formats such as HTML, JSON, text, etc,
- RESTless supports XML format.

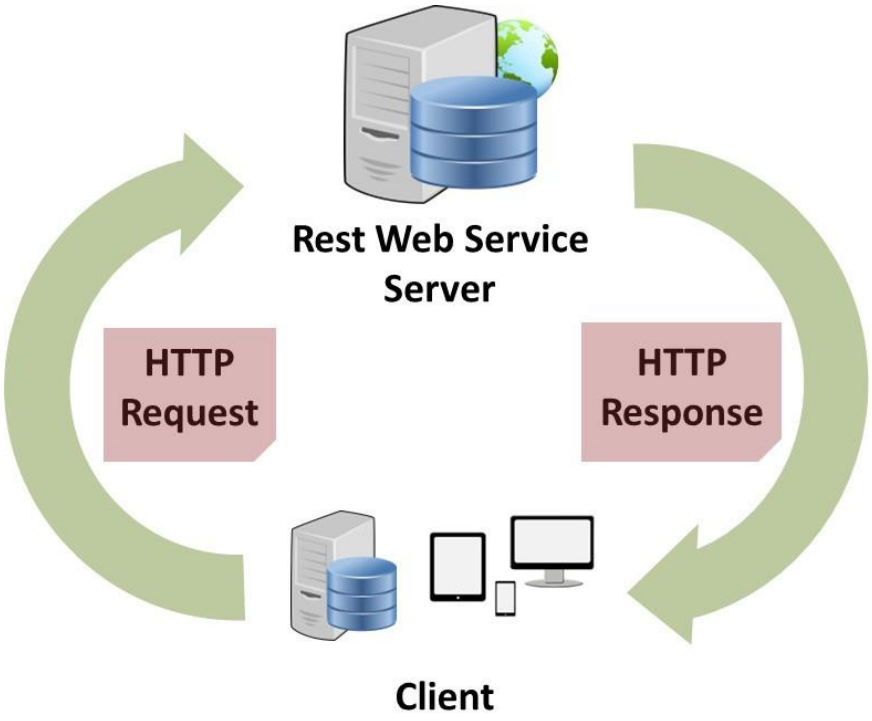
**5. Flexibility**

- RESTful is easier and flexible,
- RESTless is not as easy and flexible.

**6. Bandwidth**

- RESTful services consume less bandwidth and resource,
- RESTless services consume more bandwidth and resources.

ASP Dot NET MVC 4 is REST-Based while Microsoft WEB API is RESTFul
https://ahmetozlu93.medium.com/mastering-rest-architecture-rest-architecture-details-e47ec659f6bc




## 17. WebSocket

WebSocket is a technology that allows a client to establish two-way ("full-duplex") communication with the server. (A quick review: the client is the application on a user's computer, and the server is the remote computer that stores the website and associated data).

The key word in that definition is two-way: with WebSocket, both the client and the server can trigger communication with one another, and both can send messages, at the same time. Why is this a big deal? To fully appreciate the power of WebSocket, let's take a step back and look at a few common ways that computers can fetch data from the server.

In a traditional HTTP system, which is used by the majority of websites today, a web server is designed to receive and respond to requests from clients via HTTP messages. This traditional communication can only be initiated in one direction: from the client to the server.

- You (the client) places an order (an HTTP request) that a waiter takes to the kitchen (the server).
- The kitchen receives the order and checks if they know how to make it (the server processes the request).
- If the kitchen knows how to make the dish, they prepare the order (the server fetches data from a database or assets from the server).

- If the kitchen doesn't recognize the order or isn't allowed to serve it, they send the waiter back with bad news (if the server doesn't know how to or isn't allowed to respond to the request, it sends back an error code, like a 404).
- Either way, the waiter returns back to you (you get an HTTP response with an associated code, like 200 OK or 403 Forbidden).

The important thing to note here is that the kitchen has no idea who the order is coming from. The technical way to say this is that "HTTP is stateless": it treats each new request as completely independent. We do have ways around that—for example, clients can send along cookies that help the server identify the client, but the HTTP messages themselves are distinct and are read and fulfilled independently.

Here's the problem: the kitchen can't send a waiter to you; it can only give the waiter a dish, or bad news, when you send the waiter over. The kitchen has no concept of you—only the orders that come in. In server-speak, the only way for clients to get updated information from the server is to send requests.

Imagine a chat app where you're talking to a friend. You send a message to the server, as a request with some text as a payload. The server receives your request and stores the message. But, it has no way to reach out to your friend's computer. Your friend's computer also needs to send a request to check for new messages; only then can the server send over your message.

As it stands, you and your friend—both clients—need to constantly check the server for updates, introducing awkward delays between every message. That's silly, right? When you send a message, you want the server to ping your friend immediately to say "Hey, you got a message! Here it is!" HTTP request-response works just fine when you need to load a static page, but it's insufficient when your communication is time-sensitive.

**Short polling**

One dead simple solution to this problem is a technique called short polling. Just have the client ping the server repeatedly, say, every 500ms (or over some fixed delay). That way, you get new data every 500ms. There are a few obvious downsides to this: there's a 500ms delay, it consumes server resources with a barrage of requests, and most requests will return empty if the data isn't frequently updated.

**Long polling**

Another workaround to the delay in receiving data is a technique called long polling. In this method, the server receives a request, but doesn't respond to it until it gets new data from another request. Long polling is more efficient than pinging the server repeatedly since it saves the hassle of parsing request headers, querying for new data, and sending often-empty responses. However, the server must now keep track of multiple requests and their order. Also, requests can time out, and new requests need to be issued periodically.

**WebSockets**

So, we need a way to send information to the server, and receive updates from the server when updates come in. This brings us back to the two-way ("full-duplex") communication we mentioned earlier. Enter WebSocket! Supported by almost all modern browsers, the WebSocket API allows us to open exactly that kind of two-way connection with the server. Moreover, the server can keep track of each client and push messages to a subset of clients. Great! With this capability we can invite all of our friends to our chat app and send messages to all of them, some of them, or only your best friend.

So, how exactly does this magic work? Don't be intimidated by the setup—modern WebSocket libraries like socket.io abstract away much of the setup, but it's still helpful to understand how the technology works. If, at the end of this section, you're interested in even more detail, check out the surprisingly readable WebSocket RFC.

In order to establish a WebSocket connection with the server, the client first sends an HTTP "handshake" request with an upgrade header, specifying that the client wishes to establish a WebSocket connection. The request is sent to a ws: or wss:: URI (analogous to http or https). If the server is capable of establishing a WebSocket connection and the connection is allowed (for example, if the request comes from an authenticated or whitelisted client), the server sends a successful handshake response, indicated by HTTP code 101 Switching Protocols.

Once the connection is upgraded, the protocol switches from HTTP to WebSocket, and while packets are still sent over TCP, the communication now conforms to the WebSocket message format. Since TCP, the underlying protocol that transmits data packets, is a full-duplex protocol, both the client and the server can send messages at the same time. Messages can be fragmented, so it's possible to send a huge message without declaring the size beforehand. In that case, WebSockets breaks it up into frames. Each frame contains a small header that indicates the length and type of payload and whether this is the final frame.

A server can open WebSocket connections with multiple clients—even multiple connections with the same client. It can then message one, some, or all of these clients. Practically, this means multiple people can connect to our chat app, and we can message some of them at a time.

Finally, when it's ready to close the connection, either the client or the server can send over a "close" message.

## 18. Webhooks

There are two ways your apps can communicate with each other to share information: polling and webhooks. As one of our customer champion's friends has explained it: Polling is like knocking on your friend's door and asking if they have any sugar (aka information), but you have to go and ask for it every time you want it. Webhooks are like someone tossing a bag of sugar at your house whenever they buy some. You don't have to ask, they just automatically punt it over every time it's available.

Webhooks are automated messages sent from apps when something happens. They have a message—or payload—and are sent to a unique URL—essentially the app's phone number or address. Webhooks are almost always faster than polling, and require less work on your end.

They're much like SMS notifications. Say your bank sends you an SMS when you make a new purchase. You already told the bank your phone number, so they knew where to send the message. They type out "You just spent $10 at NewStore" and send it to your phone number +1-234-567-8900. Something happened at your bank, and you got a message about it. All is well.

Webhooks work the same way.

https://yourapp.com/data/12345?Customer=bob&value=10.00&item=paper

---

To: yourapp.com/data/12345
Message: Customer: Bob
Value: 10.00
Item: Paper