Express.js

What is Express?

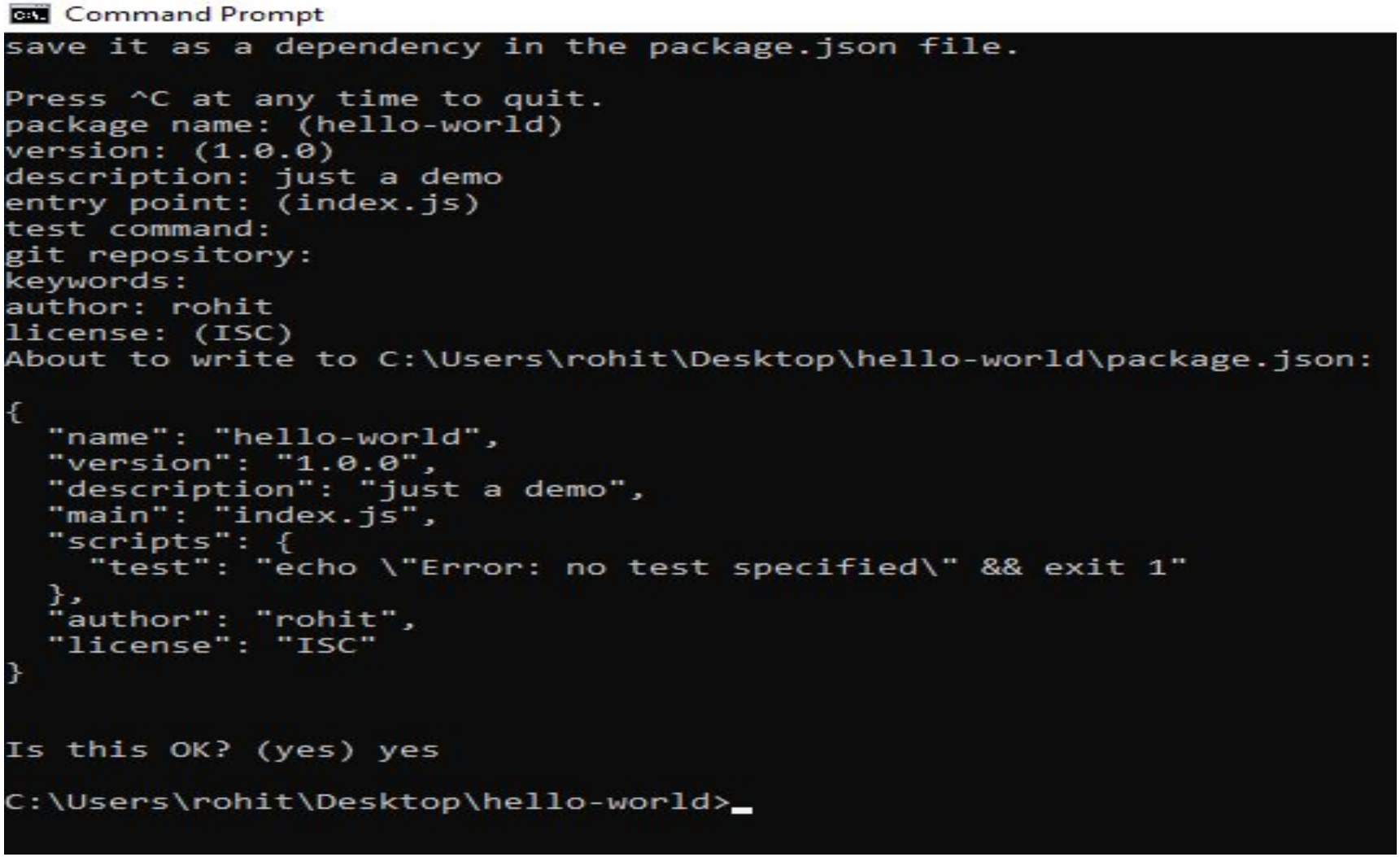
- Express.js, or simply Express, is a web application framework for Node.js, released as free and open-source software under the MIT License. It is designed for building web applications and APIs. It has been called the de facto standard server framework for Node.js
- Express is a Node.js web framework for rapid development of web applications. It provides an easy way to handle routing of an application by exposing REST APIs.

- Express is a third-party package, which means that it should be installed before importing it into your app.
- There are over 200,000 npm libraries that were developed by the Node.js. community.
- To install an npm library, run the **npm install** command.
- Whenever we create a project using npm, we need to provide a **package.json** file, which has all the details about our project. npm makes it easy for us to set up this file. Let us set up our development project.

- **Step 1** – Start your terminal/cmd, Go to Desktop and create a new folder named hello-world and cd into it –

```
C:\Users\rohit>cd Desktop
C:\Users\rohit\Desktop>mkdir hello-world
C:\Users\rohit\Desktop>cd hello-world
C:\Users\rohit\Desktop\hello-world>
```

- **Step 2** – Now to create the package.json file using npm, use the following code.
- npm init
- It will ask you for the following information.



- Just keep pressing enter, and enter your name at the “author name” field.

- **Step 3** – Now we have our package.json file set up, we will further install Express. To install Express and add it to our package.json file, use the following command –
- C:\Users\rohit\Desktop\hello-world>**npm install express --save**
- To confirm that Express has installed correctly, run the following code.
- C:\Users\rohit\Desktop\hello-world>**dir node_modules**

- **Note –**
- The --**save** flag can be replaced by the **-S** flag. This flag ensures that Express is added as a dependency to our **package.json** file. This has an advantage, the next time we need to install all the dependencies of our project we can just run the command *npm install* and it will find the dependencies in this file and install them for us.

- To make our development process a lot easier, we will install a tool from npm, **nodemon**. This tool restarts our server as soon as we make a change in any of our files, otherwise we need to restart the server manually after each file modification. To install **nodemon**, use the following command −
- C:\Users\rohit\Desktop\hello-world>**npm install -g nodemon**
- -g flag is used to make package available globally.
- You can now start working on Express.

Express.JS - Hello World Program

- Create a new file called **index.js** and type the following in it.

```
var express = require('express');
var app = express();

app.get('/', function(req, res){
  res.send("Hello world!");
});

app.listen(3000);
```

- Save the file (in hello-world folder on your desktop), go to your terminal and type the following.
- **nodemon index.js** or node index.js (if you do not want to use nodemon)
- This will start the server. To test this app, open your browser and go to **http://localhost:3000**

- **How the App Works?**
- The first line imports Express in our file, we have access to it through the variable Express. We use it to create an application and assign it to var app.
- **app.get(route, callback)**
- This function tells what to do when a **get** request at the given route is called. The callback function has 2 parameters, ***request(req)*** and ***response(res)***. The request **object(req)** represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, etc. Similarly, the response object represents the HTTP response that the Express app sends when it receives an HTTP request.

- **res.send()**
- This function takes an object as input and it sends this to the requesting client. Here we are sending the string *"Hello World!"*.
- **app.listen(port, [host], [backlog], [callback]])**
- This function binds and listens for connections on the specified host and port. Port is the only required parameter here.

S.No.	Argument & Description
1	**port** A port number on which the server should accept incoming requests.
2	**host** Name of the domain. You need to set it when you deploy your apps to the cloud.
3	**backlog** The maximum number of queued pending connections. The default is 511.
4	**callback** An asynchronous function that is called when the server starts listening for requests.

Express.JS - Routing

- Web frameworks provide resources such as HTML pages, scripts, images, etc. at different routes.
- The following function is used to define routes in an Express application −
- **app.method(path, handler)**
- This METHOD can be applied to any one of the HTTP verbs – get, set, put, delete. An alternate method also exists, which executes independent of the request type.
- Path is the route at which the request will run.
- Handler is a callback function that executes when a matching request type is found on the relevant route. For example,

```
var express = require('express');
var app = express();

app.get('/hello', function(req, res){
  res.send("Hello World!");
});

app.listen(3000);
```

- If we run our application and go to **localhost:3000/hello**, the server receives a get request at route **"/hello"**, our Express app executes the **callback** function attached to this route and sends **"Hello World!"** as the response.

- We can also have multiple different methods at the same route. For example –

```
var express = require('express');
var app = express();

app.get('/hello', function(req, res){
   res.send("Hello World!");
});

app.post('/hello', function(req, res){
   res.send("You just called the post method at '/hello'!\n");
});

app.listen(3000);
```

- To test this request, open up your terminal and run the file using nodemon or node as follows –
- C:\Users\rohit\Desktop\hello-world>**nodemon index**
- Now open second terminal and run the following command –
- C:\Users\rohit>**curl -X POST "http://localhost:3000/hello"**
- **O/P-** You just called the post method at '/hello'!

- A special method, ***all***, is provided by Express to handle all types of http methods at a particular route using the same function. To use this method, try the following.
- app.all('/test', function(req, res)
- { res.send("HTTP method doesn't have any effect on this route!"); });
- This method is generally used for defining middleware, which we'll discuss later.

- **Routers -**
- Defining routes like above is very tedious to maintain. To separate the routes from our main **index.js** file, we will use **express.Router**. Create a new file called **things.js** and type the following in it.

```
var express = require('express');
var router = express.Router();

router.get('/', function(req, res){
  res.send('GET route on things.');
});
router.post('/', function(req, res){
  res.send('POST route on things.');
});

//export this router to use in our index.js
module.exports = router;
```

- Now to use this router in our **index.js**, type in the following before the **app.listen** function call –

```
var express = require('Express');
var app = express();

var things = require('./things.js');

//both index.js and things.js should be in same directory
app.use('/things', things);

app.listen(3000);
```

- The ***app.use*** function call on route **'/things'** attaches the **things** router with this route. Now whatever requests our app gets at the '/things', will be handled by our things.js router. The **'/'** route in things.js is actually a subroute of '/things'.
- Visit localhost:3000/things/

- Routers are very helpful in separating concerns and keep relevant portions of our code together. They help in building maintainable code. You should define your routes relating to an entity in a single file and include it using the above method in your **index.js** file.

Express.JS - URL Building

- We can now define routes, but those are static or fixed. To use the dynamic routes, we SHOULD provide different types of routes. Using dynamic routes allows us to pass parameters and process based on them.
- Here is an example of a dynamic route −

```
var express = require('express');
var app = express();

app.get('/:id', function(req, res){
   res.send('The id you specified is ' + req.params.id);
});
app.listen(3000);
```

- To test this go to **http://localhost:3000/123**.

- You can replace '123' in the URL with anything else and the change will reflect in the response. A more complex example of the above is −

```
var express = require('express');
var app = express();

app.get('/things/:name/:id', function(req, res) {
  res.send('id: ' + req.params.id + ' and name: ' + req.params.name);
});
app.listen(3000);
```

- To test the above code, go to **http://localhost:3000/things/Ajay/12345**.

- You can use the ***req.params*** object to access all the parameters you pass in the url.

Express.JS - Middleware

- Middleware functions are functions that have access to the **request object (req)**, the **response object (res)**, and the next middleware function in the application's request-response cycle. These functions are used to modify **req** and **res** objects for tasks like parsing request bodies, adding response headers, etc.
- Here is a simple example of a middleware function in action −

```
var express = require('express');
var app = express();

//Simple request time logger
app.use(function(req, res, next){
  console.log("A new request received at " + Date.now());

  //This function call is very important. It tells that more processing is
  //required for the current request and is in the next middleware
  //function/route handler.
  next();
});

app.get('/', function(req, res){
  res.send('This is just a demo');

});
app.listen(3000);
```

- The above middleware is called for every request on the server. So after every request, we will get the following message in the console −
- A new request received at 1581920483624
- And whatever will be the output of request handler function will be shown in browser.
- To restrict it to a specific route (and all its subroutes), provide that route as the first argument of ***app.use()***. For Example -

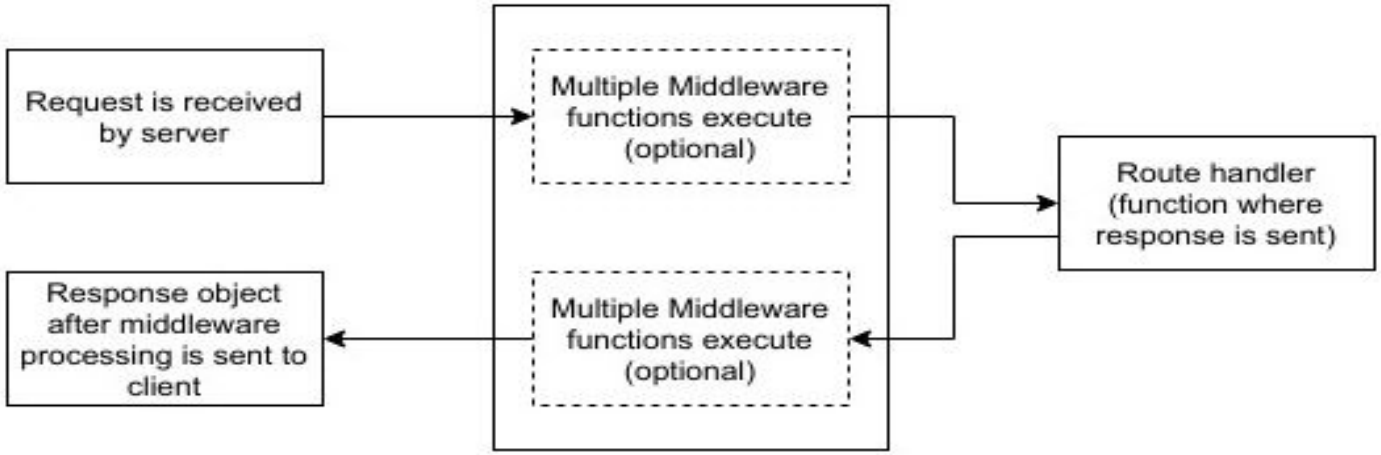
```
var express = require('express');
var app = express();

//Middleware function to log request protocol
app.use('/things', function(req, res, next){
  console.log("A request for things received at " + Date.now());
  next();
});

// Route handler that sends the response
app.get('/things', function(req, res){
  res.send('Things');
});

app.listen(3000);
```

- Now whenever you request any subroute of '/things', only then it will log the time.

- **Order of Middleware Calls**
- One of the most important things about middleware in Express is the order in which they are written/included in your file; the order in which they are executed, given that the route matches also needs to be considered.
- For example, in the following code snippet, the first function executes first, then the route handler and then the end function. This example summarizes how to use middleware before and after route handler; also how a route handler can be used as a middleware itself.

```
var express = require('express');
var app = express();

//First middleware before response is sent
app.use(function(req, res, next){
  console.log("Start");
  next();
});

//Route handler
app.get('/', function(req, res, next){
  res.send("Middle");
  next();
});

app.use('/', function(req, res){
  console.log('End');
});

app.listen(3000);
```

- When we visit '/' after running this code, we receive the response as **Middle** and on our console −
- Start
- End
- The following diagram summarizes what we have learnt about middleware −



- **Third Party Middleware -**

- **body-parser middleware –**
- Parse incoming request bodies in a middleware before your handlers, available under the req.body property.
- **Note** As req.body's shape is based on user-controlled input, all properties and values in this object are untrusted and should be validated before trusting.
- Here we will use body-parser to extract information from a POST body in Express.js.
- The HTTP protocol provides a number of ways to pass information from a client to a server, with POST bodies being the most flexible and most commonly used method to send data via HTTP.
- Another way, which is typically used for different use-cases, is to convey information using query strings or URL parameters.

- **Sending POST Data in HTTP**
- Data can be sent via an HTTP POST call for many reasons, with some of the most common being via an HTML <form> or an API request. Content-type can have one of the following form -
- **application/x-www-form-urlencoded:** Data in this encoding is formatted like a query string you'd see in a URL, with key-value Paris being separated by & characters. For example: foo=bar&abc=123&stack=abuse. This is the default encoding.
- **multipart/form-data:** This encoding is typically used for sending files. In short, each key-value is sent in the same request, but different "parts", which are separated by "boundaries" and include more meta-data.
- **text/plain:** This data is just sent as unstructured plain text, and typically is not used.
- A raw HTTP POST request with the **application/x-www-form-urlencoded** encoding might look something like this:

```
POST /signup HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 53

username=scott&password=secret&website=stackabuse.com
```

- **Extracting POST Data**
- Before we can get started accessing POST data right away, we need to properly configure our Express app.
- To set this up, we'll be using the body-parserTo set this up, we'll be using the body-parser package, which can handle many forms of data. This package is a middleware that intercepts the raw body and parses it in to a form that your application code can easily use.
- Install body-parser as follows –
- C:\Users\rohit\Desktop\hello-world>**npm install body-parser --save**

```
// index.js
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
app.use(bodyParser.urlencoded({ extended: true }));

app.post('/post-test', (req, res) => {
    console.log('Got body:', req.body);
    res.sendStatus(200);
});

app.listen(8080, () => console.log(`Started server at
    http://localhost:8080!`));
```

- Notice how we call app.use(...) *before* defining our route. The order here matters. This will ensure that the body-parser will run before our route, which ensures that our route can then access the parsed HTTP POST body.
- To test this, we'll first start the Express app and then use the curl utility in a different console window:
- C:\Users\rohit\Desktop\hello-world>**nodemon index**

- C:\Users\rohit>**curl -d "username=scott&password=secret&website=stackabuse.com" -X POST http://localhost:8080/post-test**

- **O/p in prevous cmd –**

Started server at http://localhost:8080!
Got body: { username: 'scott',
  password: 'secret',
  website: 'stackabuse.com' }

- Here you can see that the query string data was parsed in to a JavaScript object that we can easily access.

- urlencoded is one of the most commonly used parsers that body-parser provides, you can also use the following:
- .json(): Parses JSON-formatted text for bodies with a Content-Type of application/json.
- .raw(): Parses HTTP body in to a Buffer for specified custom Content-Types, although the default
  accepted Content-Type is application/octet-str eam.
- .text(): Parses HTTP bodies with
  a Content-Type of text/plain, which returns it as a plain string.

- The great thing about the middleware model and how this package parses data is that you're not stuck to using just one parser. You can enable one *or more* parsers for your app to ensure that all data types are processed properly:

```
// index.js
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
app.use(bodyParser.raw());

// ...
```

- So now if we were to send an HTTP POST request with JSON as the body, it will be parsed in to a regular JSON object in the req.body property:

```
POST /post-test HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 69

'{"username":"scott","password":"secret","website":"stackabuse.com"
    }‘


$ node index.js
Started server at http://localhost:8080!
Got body: { username: 'scott',
  password: 'secret',
  website: 'stackabuse.com' }
```

- **Conclusion**
- The most common way to send diverse and large amounts of data via HTTP is to use the POST method. Before we can easily access this data on the server side in Express, we need to use some middleware, like
  the body-parser package, to parse the data in to a format that we can easily access. Once the data from the raw HTTP request is parsed, it can then be accessed via the body property of the req object.

ExpressJS - Templating (PUG.js)

- Pug is a templating engine for Express. you can write much simpler Pug code, which Pug compiler will compile into HTML code, that browser can understand.
- But why not write HTML code in the first place?
- Pug has powerful features like conditions, loops, includes, mixins, using which we can render HTML code based on user input or reference data. Pug also support JavaScript natively, hence using JavaScript expressions, we can format HTML code.

- To use Pug with Express, we need to install it,
- **npm install --save pug**
- Now that Pug is installed, set it as the templating engine for your app.
  You **don't** need to 'require' it. Add the following code to your **index.js** file.
- **app.set('view engine', 'pug');**
  app.set('views','./views');
- Now create a new directory called views.
  Inside that create a file called **first_view.pug**, and enter the following data in it.

```
doctype html
html
  head
    title = "Hello Pug"
  body
    p.greetings#people Hello World!
```

- To run this page, add the following route to your app −

```
var express = require('express');
var app = express();
app.set('view engine', 'pug');
app.set('views','./views');

app.get('/first_template', function(req, res){
  res.render('first_view');
});

app.listen(3000);
```

- Now run it using nodemon and enter the following url in browser –
- http://localhost:3000/first_template
- You will get the output as − **Hello World!** Pug converts this very simple looking markup to html. We don't need to keep track of closing our tags, no need to use class and id keywords, rather use '.' and **'#'** to define them. The above code first gets converted to −

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello Pug</title>
  </head>

  <body>
    <p class = "greetings" id = "people">Hello World!</p>
  </body>
</html>
```

- Pug is capable of doing much more than simplifying HTML markup.
- **Important Features of Pug -**
- Let us now explore a few important features of Pug.
- Simple Tags
- Tags are nested according to their indentation. Like in the above example, **<title>** was indented within the **<head>** tag, so it was inside it. But the **<body>** tag was on the same indentation, so it was a sibling of the **<head>** tag.
- We don’t need to close tags, as soon as Pug encounters the next tag on same or outer indentation level, it closes the tag for us.

- To put text inside of a tag, we have 3 methods –
- **Space seperated -**
- h1 Welcome to Pug
- **Piped text -**
- div
- | To insert multiline text,
  | You can use the pipe operator.
- **Block of text -**
- div.
- But that gets tedious if you have a lot of text.
  You can use "." at the end of tag to denote block of text. To put tags inside this block, simply enter tag in a new line and indent it accordingly .

- **Comments**
- Pug uses the same syntax as **JavaScript(//)** for creating comments. These comments are converted to the html comments(<!--comment-->). For example,
- //This is a Pug comment
- This comment gets converted to the following.
- <!--This is a Pug comment-->

- **Attributes**
- To define attributes, we use a comma separated list of attributes, in parenthesis. Class and ID attributes have special representations. The following line of code covers defining attributes, classes and id for a given html tag.
- div.container#division(width = "100", height = "100")

- This line of code, gets converted to the following. −
-
- **Passing Values to Templates**
- When we render a Pug template, we can actually pass it a value from our route handler, which we can then use in our template. Create a new route handler with the following.

```
var express = require('express');
var app = express();
app.set('view engine', 'pug');
app.set('views','./views');

app.get('/dynamic_view', function(req, res){
   res.render('dynamic', {
      name: "Demo",
      url:"http://www.demo.com"
   });
});

app.listen(3000);
```

- And create a new view file in views directory, called **dynamic.pug**, with the following code –

```
html
  head
    title=name
  body
    h1=name
    a(href = url) URL
```

- We can also use these passed variables within text. To insert passed variables in between text of a tag, we use **#{variableName}** syntax. For example, in the above example, if we wanted to put Greetings from Demowebsite, then we could have done the following.

```
html
  head
    title = name
  body
    h1 Greetings from #{name}
    a(href = url) URL
```

- **Conditionals**
- We can use conditional statements and looping constructs as well.
- Consider the following −
- If a User is logged in, the page should display **"Hi, User"** and if not, then the **"Login/Sign Up"** link. To achieve this, we can define a simple template like −

```
html
  head
    title Simple template
  body
    if(user)
      h1 Hi, #{user.name}
    else
      a(href = "/sign_up") Sign Up
```

- When we render this using our routes, we can pass an object as in the following program −

```
var express = require('express');
var app = express();
app.set('view engine', 'pug');
app.set('views','./views');

app.get('/dynamic_view', function(req, res){
   res.render('dynamic', {
   user: {name: "Ayush", age: "20"}
   });
});

app.listen(3000);
```

- You will receive a message − **Hi, Ayush**. But if we don’t pass any object or pass one with no user key, then we will get a signup link.

- **Include and Components**
- Pug provides a very intuitive way to create components for a web page. For example, if you see a news website, the header with logo and categories is always fixed. Instead of copying that to every view we create, we can use the **include** feature. Following example shows how we can use this feature −
- Create 3 views with the following code −
- **HEADER.PUG –**

```
div.header.
  I'm the header for this website.
```

- **FOOTER.PUG –**

```
div.footer.
  I'm the footer for this website.
```

- **CONTENT.PUG –**

```
html
  head
    title Simple template
  body
    include ./header.pug
    h3 I'm the main content
    include ./footer.pug
```

- Create a route for this as follows −

```
var express = require('express');
var app = express();
app.set('view engine', 'pug');
app.set('views','./views');

app.get('/components', function(req, res){
   res.render('content');
});

app.listen(3000);
```

- ***include*** can also be used to include plaintext, css and JavaScript.
- There are many more features of Pug. But those are out of the scope for this tutorial. You can further explore Pug