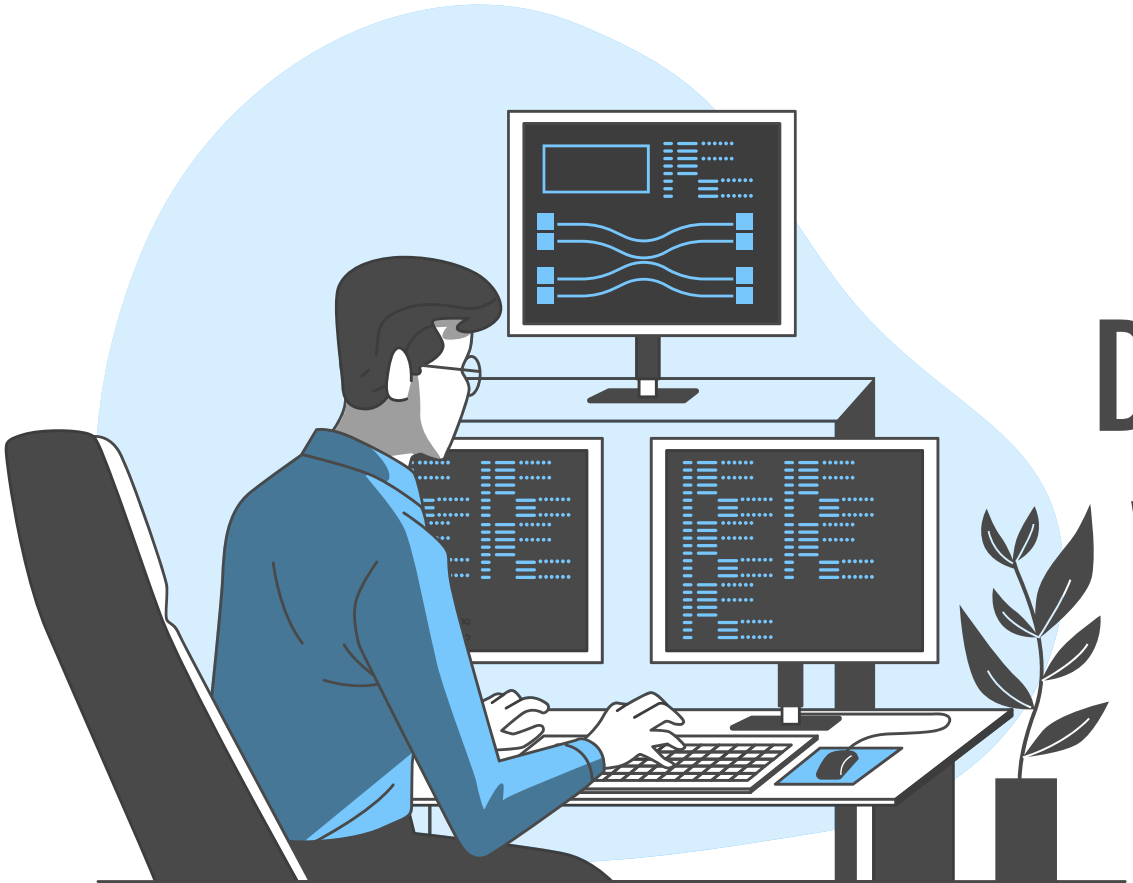


Back-End Development with NodeJS

Prepared by Jawad Srour

What is NodeJS

What is NodeJS



Runtime Environment

A JavaScript run-time environment used commonly for building back-end services



Advantages

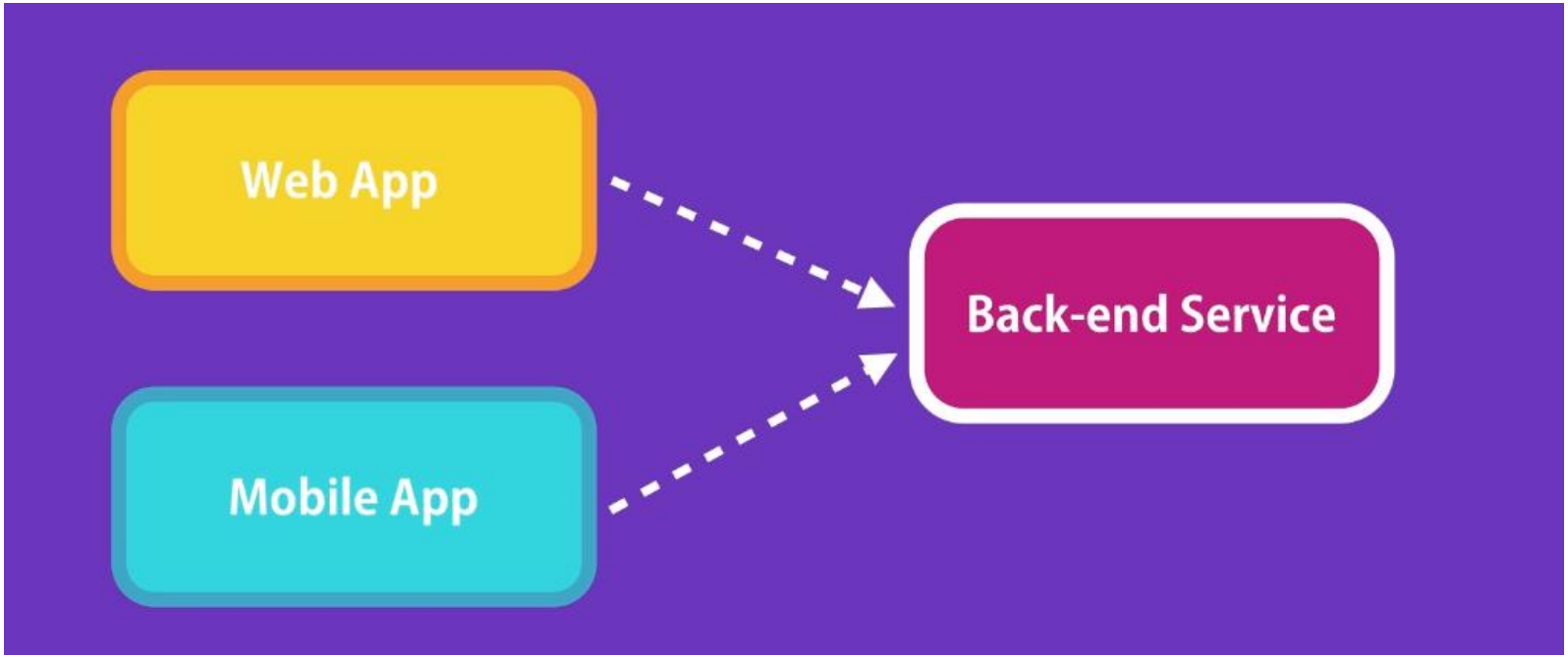
Highly-scalable, data-intensive an real-time apps. It is ideal for I/O-intensive apps



Fast Development

Great for prototyping ang Agile Development

Back-end Services

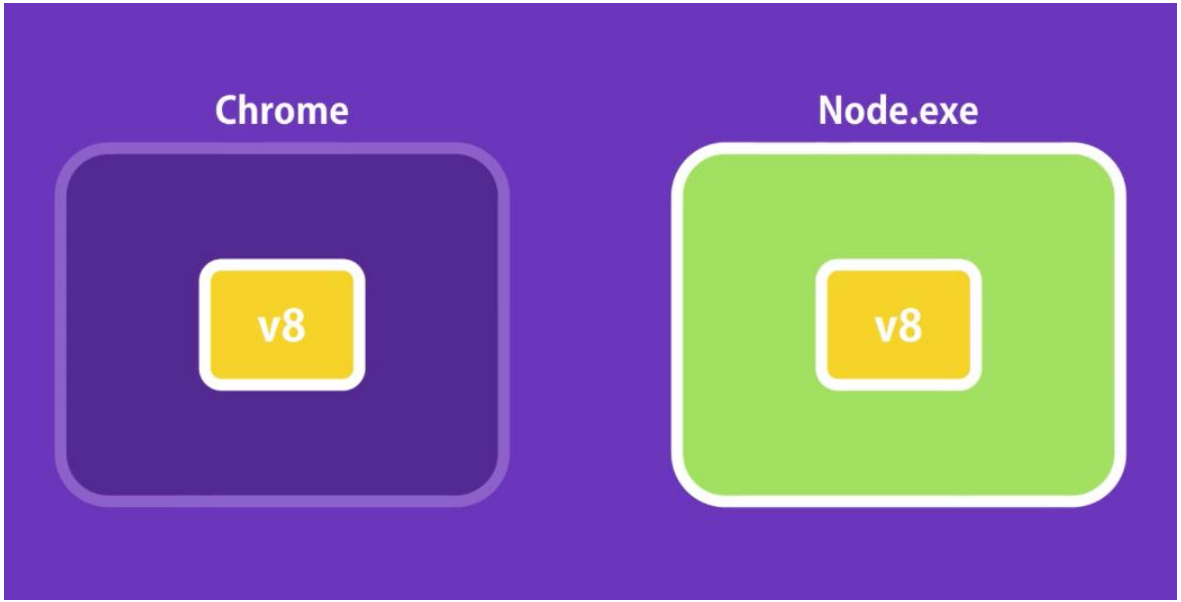


NodeJS Is not a Framework

It's a **runtime environment**

for executing JavaScript code

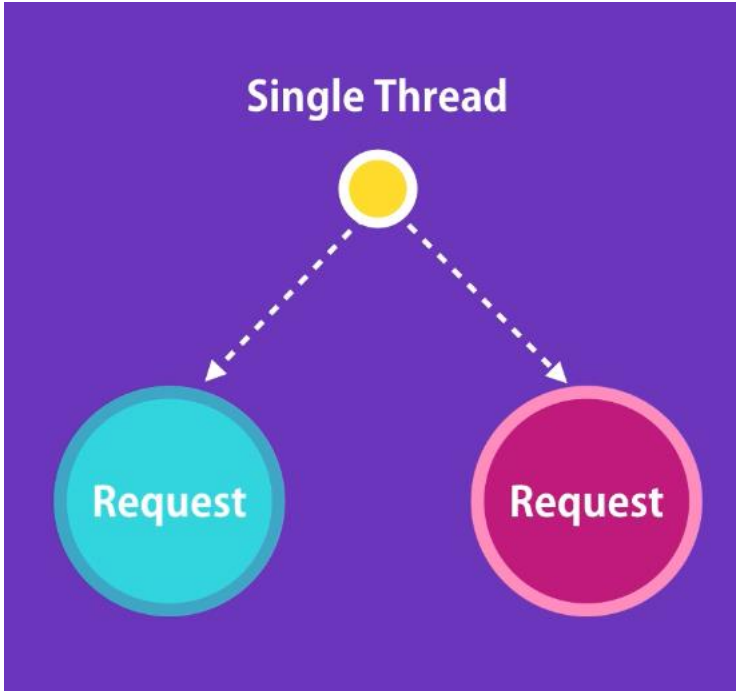
NodeJS Architecture



1. It is a C++ Program that contains Chrome's JavaScript V8 Engine
2. Having the V8 Engine, it allows executing JavaScript code outside a browser
3. Allows working with the File System or the network for example.

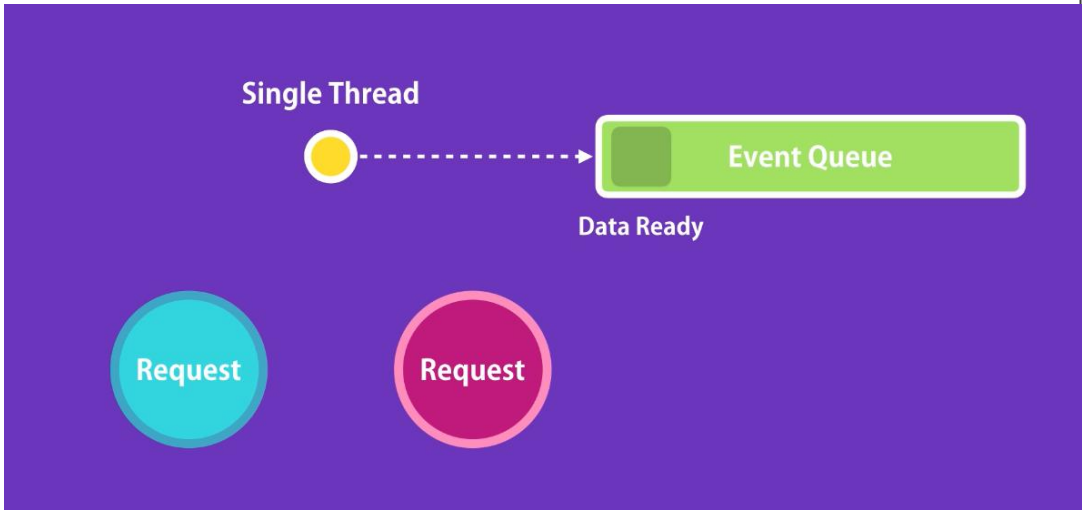
How Node Works

- Non-Blocking Asynchronous Architecture
- A single thread is used to handle multiple requests.
- Note: JavaScript is single-threaded
- Example: Chef takes multiple orders instead of customers waiting for each meal to be done before ordering theirs.



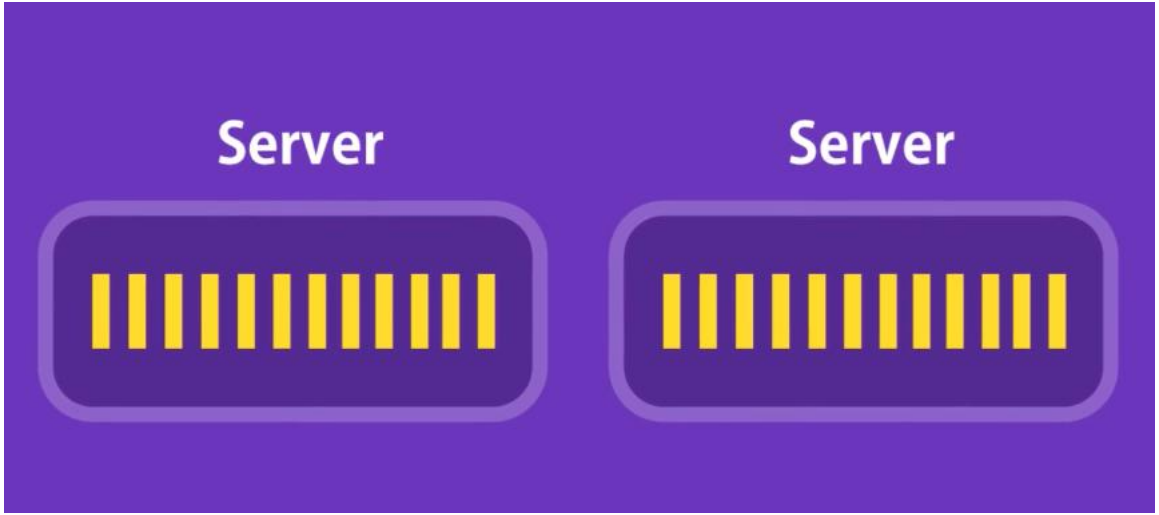
How Node Works

- Non-Blocking Asynchronous Architecture
- A single thread is used to handle multiple requests.
- Note: JavaScript is single-threaded
- Example: Chef takes multiple orders instead of customers waiting for each meal to be done before ordering theirs.



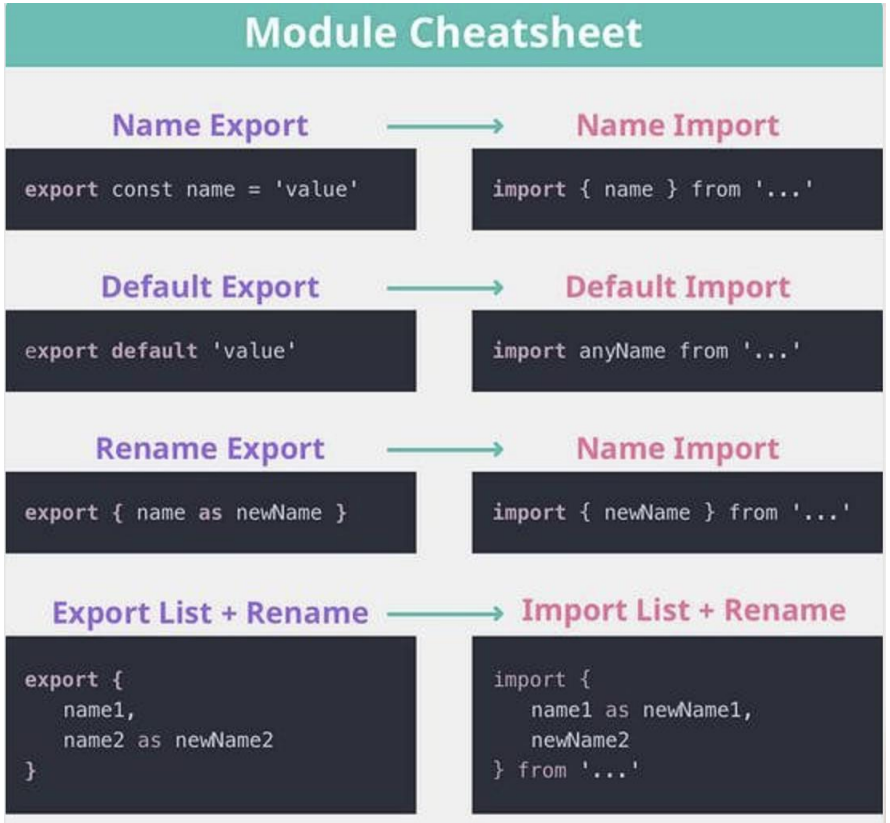
Issue with Blocking Synchronous Architecture

- We spawn a thread for each new request (user)
- We will eventually run out of threads, or things will get costly.
- In-efficient use of resources
- ASP .NET is blocking synchronous by default.
- Node is async by default.



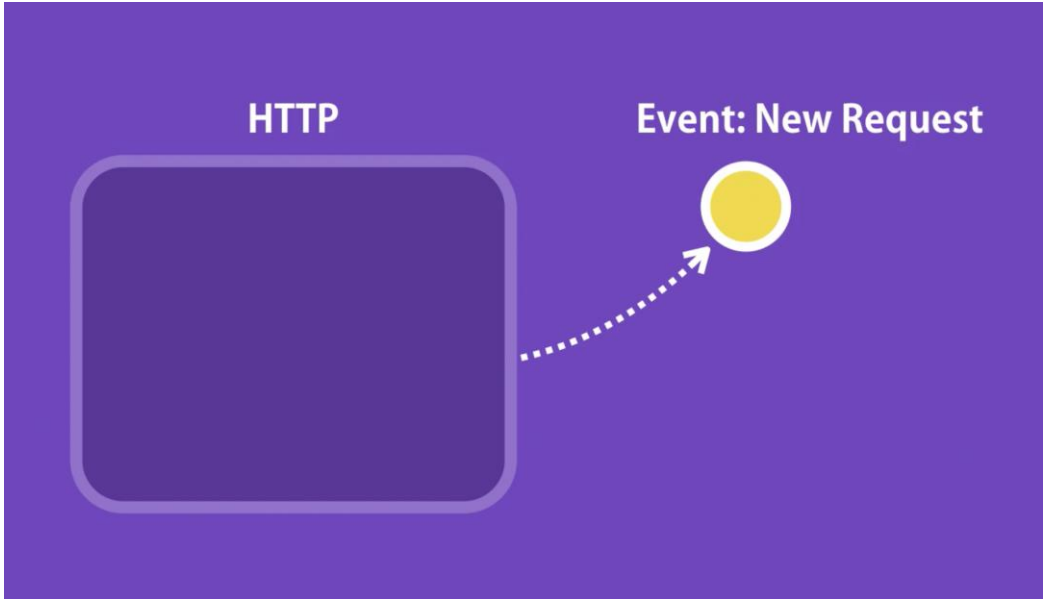
Modules

- 1 JS file == 1 module
- We export modules when we need their functionality in other files
- Node's modularity structure is useful to avoid having functions and objects un-intentionally overwritten.
- Functions, objects, and variables are file-private by default.

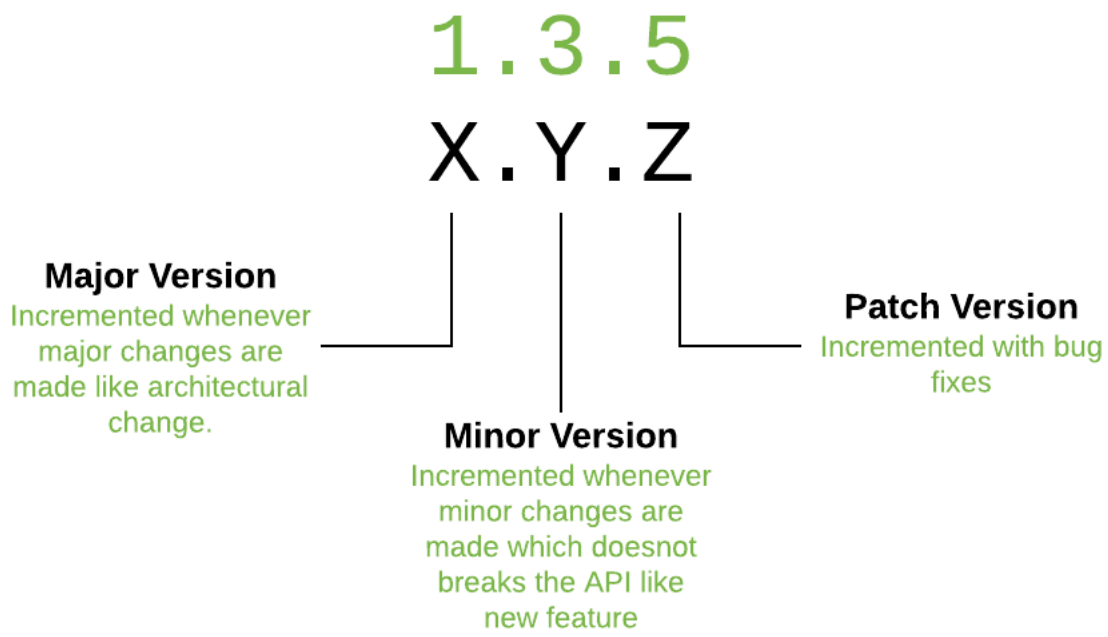


Events

- Event: A signal that something has happened
- A request is received, event (new request)
- We respond to that event with the response.
- EventEmitter → event gets emitted
- EventHandler → event gets handled
- Equivalent of Signals/Slots in C++ Qt

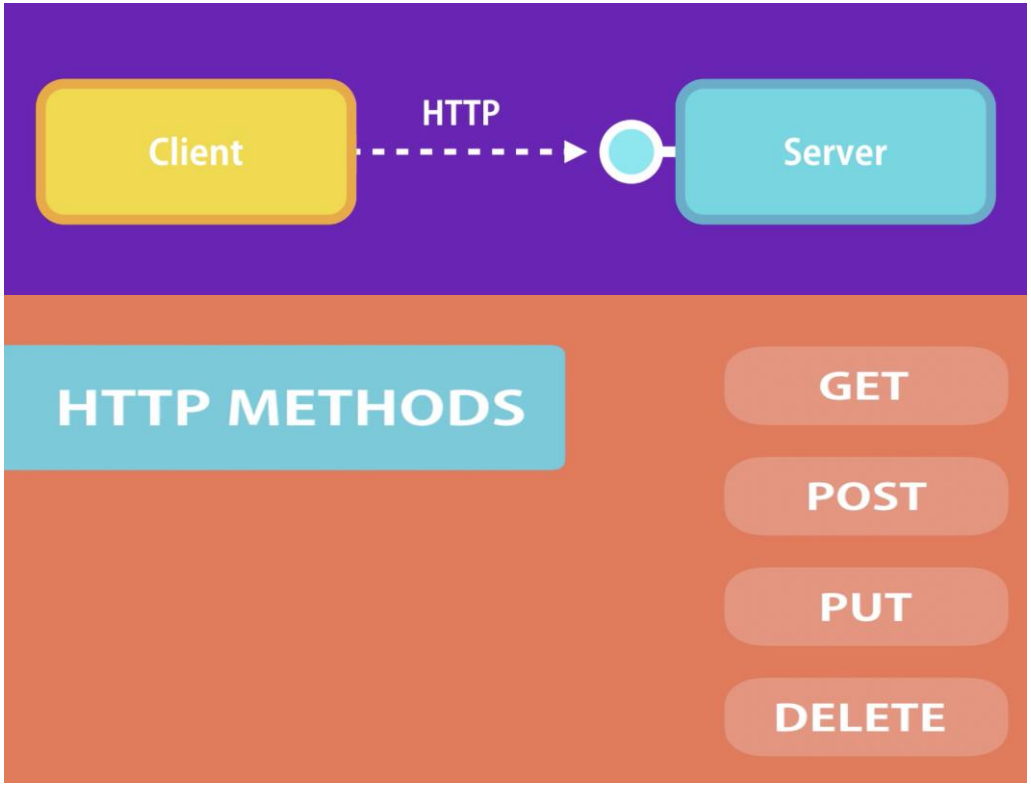


Semantic Versioning



RESTful Services

- Expose API end-points for users.
- Communication is done using the HTTP protocol.
- REST: Representational State Transfer
- REST is a convention for building the HTTP services.
- HTTP Verbs: GET, POST, PUT, DELETE



Environment Variables

Abstract Data

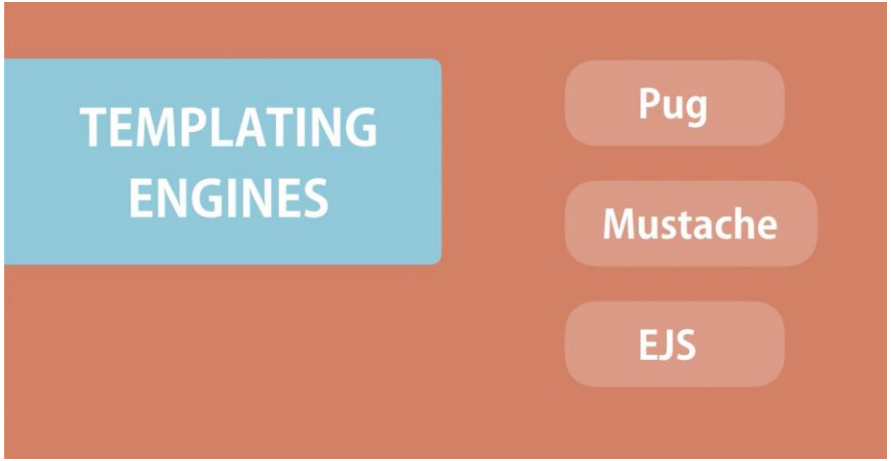
Abstract from the user the information they don't need to know about

Security

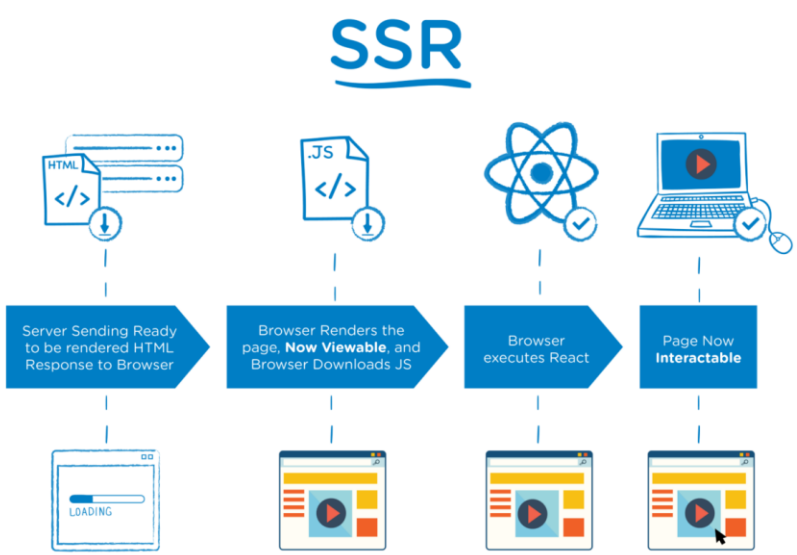
We don't want to push to a remote repository our admin credentials. (example: DBMS account credentials)

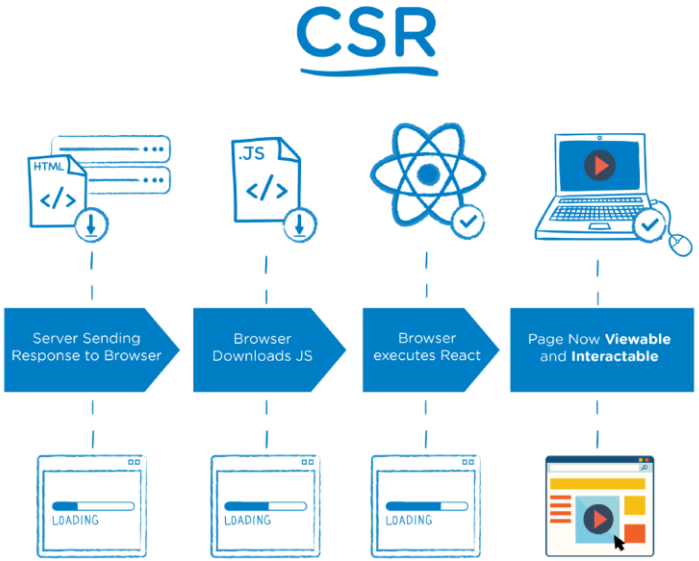
Templating Engines

- We can do Server-Side Rendering with Templating Engines.
- HTML pages are rendered from the server
- The templating engines allow embedding data sent from the back-end into the HTML pages before they get rendered.
- Might be faster for small-scale apps.



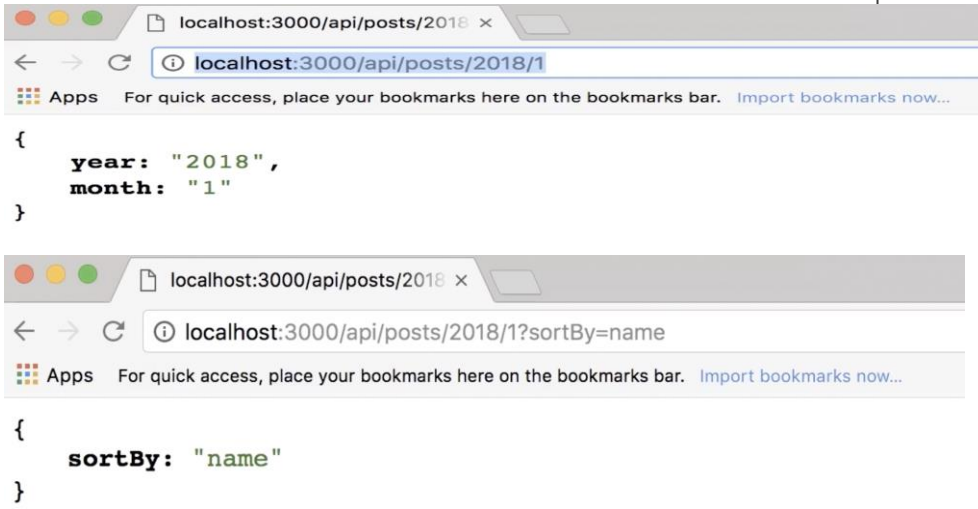
Server-Side Rendering VS Client-Side Rendering





Path(Route) Parameters VS Query Parameters

- Use Path Parameters for Essential/Required Information.
- Use Query String Parameters for anything that is optional (providing further instructions to the server such as sorting the customers by name)



Joi NPM Package For Input Validation

Write Less Validation Code

Instead of manually writing validation logic, we can use Joi for validating user input

```
function validateCourse(course) {
  const schema = {
    name: Joi.string().min(3).required()
  };

  return Joi.validate(course, schema);
}

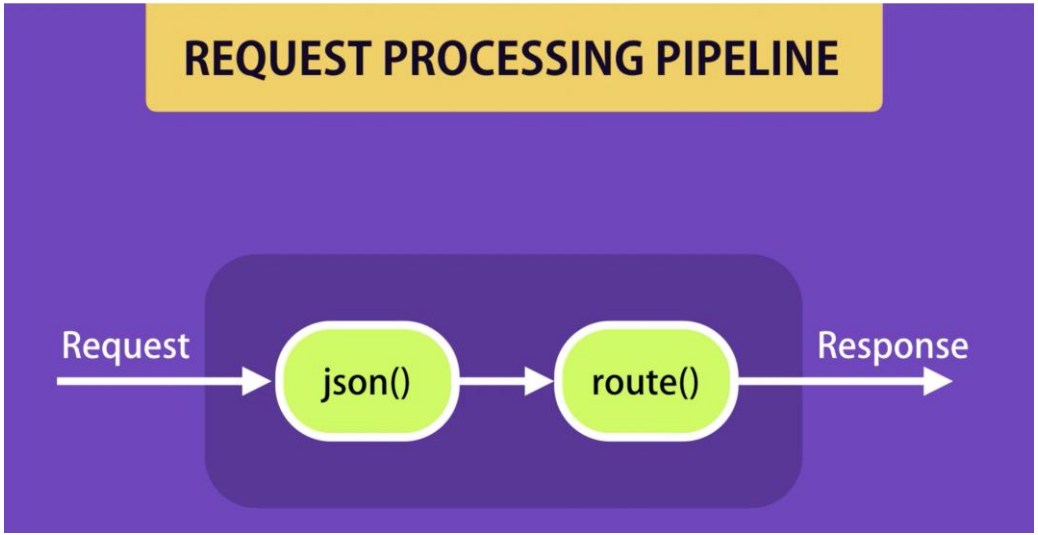
router.post('/', (req, res) => {
  const { error } = validateCourse(req.body);
  if (error) return res.status(400).send(error.details[0].message);

  const course = {
    id: courses.length + 1,
    name: req.body.name
  };
  courses.push(course);
  res.send(course);
});
```

Middleware

A third party providing my application with services not offered by the OS. example: openCV

Note: Firmware, unlike middleware, is software that allows controlling a device's hardware



Pagination

- Often, we need Pages in our applications
- To implement pagination on the back-end side, when the user requests a page, we retrieve the corresponding records from the database for that page.
- We need to define 2 things:
  - Page Size
  - Page Number

```
async function getCourses() {
  const pageNumber = 2;
  const pageSize = 10;

  const courses = await Course
    .find({ author: 'Mosh', isPublished: true })
    .skip((pageNumber - 1) * pageSize)
    .limit(pageSize)
    .sort({ name: 1 })
    .select({ name: 1, tags: 1 });
  console.log(courses);
}
```

Asynchronous JavaScript

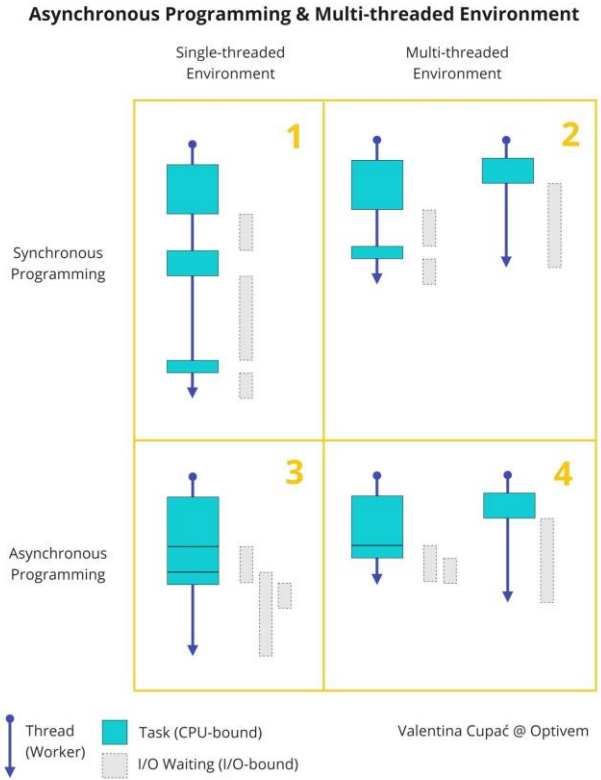
Asynchronous JavaScript

- JavaScript is single-threaded.
- Asynchronous does not mean multi-threaded.
- Asynchronous means we are not blocked by I/O operations. The program can proceed doing its tasks until the asynchronous operation returns
- Synchronous means line by line execution.

```
1  console.log("First line executed")
2  setTimeout( () => {
3      console.log("Second line executed")
4  }, 2000);
5  console.log("Third line executed")
6  console.log("Fourth line executed")
7
```

Asynchronous JavaScript

1. Synchronous single-threaded: blocking / sequential
2. Synchronous multi-threaded: still blocking + sequential, but we have more than 1 worker to execute the required tasks → faster
3. Asynchronous single-threaded: non-blocking, but 1 worker
4. Asynchronous multi-threaded: non-blocking + more than 1 worker



Callback

- Callback: a function that gets executed after an asynchronous operation completes.
- A function that gets called back in the body of another function.
- Callback hell: nested anonymous function calls.
- Fix to callback hell => Named functions.

```
getCustomer(1, (customer) => {
  console.log('Customer: ', customer);
  if (customer.isGold) {
    getTopMovies((movies) => {
      console.log('Top movies: ', movies);
      sendEmail(customer.email, movies, () => {
        console.log('Email sent...')
      });
    });
  }
});

function getCustomer(id, callback) {
  setTimeout(() => {
    callback({
      id: 1,
      name: 'Mosh Hamedani',
      isGold: true,
      email: 'email'
    });
  }, 4000);
}

function getTopMovies(callback) {
  setTimeout(() => {
    callback(['movie1', 'movie2']);
  }, 4000);
}

function sendEmail(email, movies, callback) {
  setTimeout(() => {
    callback();
  }, 4000);
}
```

Callback

- Callback hell: nested anonymous function calls.
- Fix to callback hell => Named functions.
- Sounds confusing? There is a better approach...

```
getCustomer(1, getCustomerCallback);

// Callbacks
// we pass reference to these functions
function getCustomerCallback(customer){
  console.log('Customer: ', customer);
  if (customer.isGold){
    getTopMovies(getTopMoviesCallback);
  }
}

function getTopMoviesCallback(movies){
    console.log('Top movies: ', movies);
    sendEmail(customer.email, movies, sendEmailCallback);
}

function sendEmailCallback(){
  console.log('Email sent...')
}

function getCustomer(id, callback) {
  setTimeout(() => {
    callback({
      id: 1,
      name: 'Mosh Hamedani',
      isGold: true,
      email: 'email'
    });
  }, 4000);
}

function getTopMovies(callback) {
  setTimeout(() => {
    callback(['movie1', 'movie2']);
  }, 4000);
}

function sendEmail(email, movies, callback) {
  setTimeout(() => {
    callback();
  }, 4000);
}
```

Promises

- We can use Promises instead of Callbacks
- We replace the callbacks with promises
- then() is executed when the promise is resolved (fulfilled), and catch is executed when the promise is rejected (fails)
- Promise initially is in the pending state

```
// Promise-based approach
getUser(1)
  .then(user => getRepositories(user.gitHubUsername))
  .then(repos => getCommits(repos[0]))
  .then(commits => console.log('Commits', commits))
  .catch(err => console.log('Error', err.message));


console.log('After');

function getUser(id) {
  return new Promise((resolve, reject) => {
    // Kick off some async work
    setTimeout(() => {
      console.log('Reading a user from a database...');
      resolve({ id: id, gitHubUsername: 'mosh' });
    }, 2000);
  });
}

function getRepositories(username) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Calling GitHub API...');
      resolve(['repo1', 'repo2', 'repo3']);
    }, 2000);
  });
}

function getCommits(repo) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Calling GitHub API...');
      resolve(['commit']);
    }, 2000);
  });
}
```

bnine, Line 21, Column 22

Async Await

- A function that returns a promise object, can be awaited using the ‘await’ keyword.
- The function that ‘awaits’ needs to be declared as ‘async’

```
async function notifyCustomer() {
  const customer = await getCustomer(1);
  console.log('Customer: ', customer);
  if (customer.isGold) {
    const movies = await getTopMovies();
    console.log('Top movies: ', movies);
    await sendEmail(customer.email, movies);
    console.log('Email sent...');
  }
}
notifyCustomer();


function getCustomer(id) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve({
        id: 1,
        name: 'Mosh Hamedani',
        isGold: true,
        email: 'email'
      });
    }, 4000);
  });
}

function getTopMovies() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(['movie1', 'movie2']);
    }, 4000);
  });
}

function sendEmail(email, movies) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve();
    }, 4000);
  });
}
```

Async Await

- A function that returns a promise object, can be awaited using the ‘await’ keyword.
- The function that ‘awaits’ needs to be declared as ‘async’

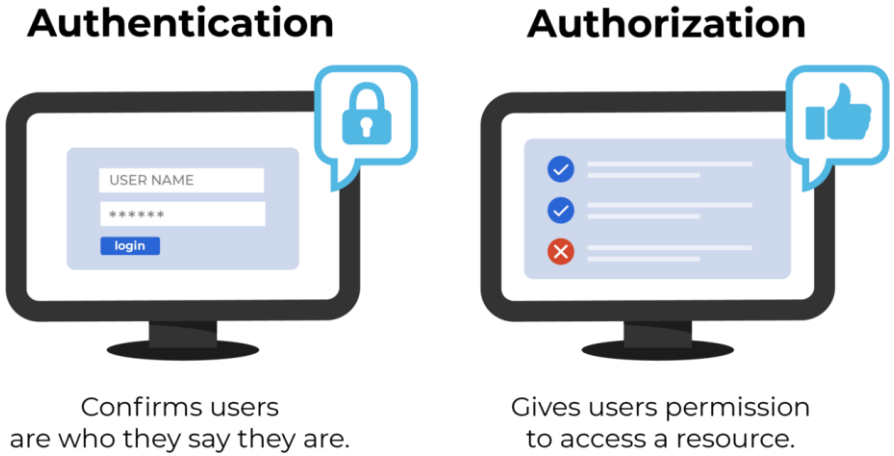
```
async function notifyCustomer() {

  try{
    const customer = await getCustomer(1);
    console.log('Customer: ', customer);
    if (customer.isGold) {
      const movies = await getTopMovies();
      console.log('Top movies: ', movies);
      await sendEmail(customer.email, movies);
      console.log('Email sent...');
    }
  }
  catch(err){
    console.log(err);
  }
}
```

Authentication and Authorization

Authentication VS Authorization

- We want to protect our endpoints.
- We don't want an anonymous user to perform transactions on our application→ we need user authentication.
- We don't want Bob to access the wallet of Alice. Only Alice should be able to access her wallet → We need authorization
- **authentication is the process of verifying who someone is, whereas authorization is the process of verifying what specific applications, files, and data a user has access to.**
- We authenticate users by persisting information about the user (login credentials)
- Authorize by setting permissions (example: Boolean to check if isAdmin)



okta

Some Common HTTP Response Status Codes

HTTP 404

Not Found.
Useful when the requested resource is not found. (User does not exist)

HTTP 500

Internal Server Error.
An error has occurred at the server.

HTTP 400

Bad Request.
There is an issue with the body of the request sent by the user.

HTTP 301

Moved Permanently.
The URL of the requested resource has been changed permanently

HTTP 401

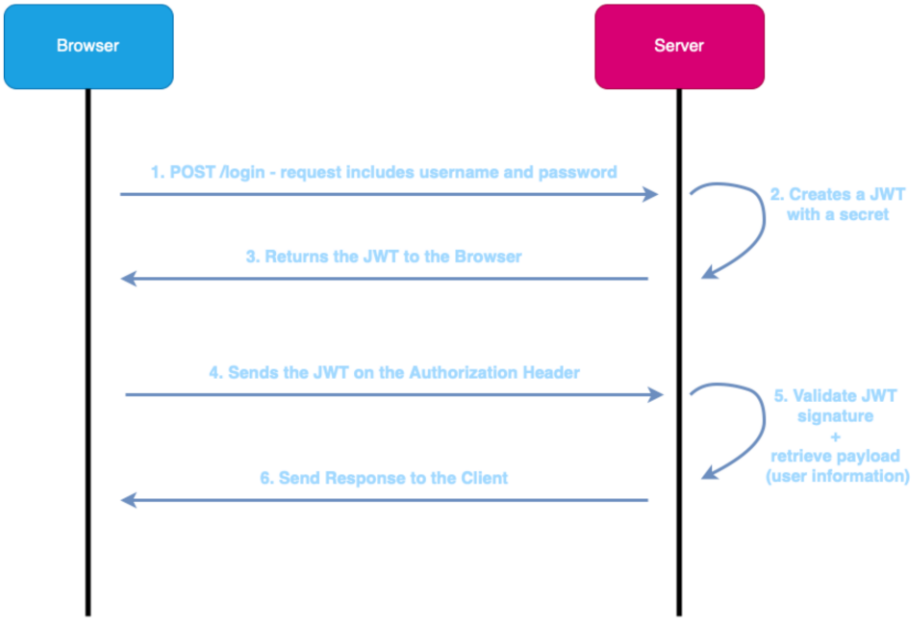
Un-authorized.
Means the client is un-authenticated

HTTP 403

Forbidden.
Client is authenticated, but no permissions to access the requested resource

JSON Web Tokens

- JSON Web Tokens, also referred to as "Bearer Token".
- A JSON web token is an encoded string that holds relevant user information, and the user gets hold of their own token on successful login.
- Then, the user needs to provide that token with future requests to API resources that require being an authenticated user (automated process from the front-end application).
- JWT expires after a set time.



Hashing Passwords

- Never store a user's password in your database as is (plain text password)
- If any database vulnerability happens, user passwords get exposed, and users typically use the same passwords in different apps ... THIS IS BAD!
- Solution? We hash our passwords!
- A hash function is any function that can be used to map data of arbitrary size to fixed-size values.
- Hash functions map a given key to what is called hashed value
- A hash function produces a unique hashed value for a given key (same key == same hashed value ... problem?)
- There are various cryptographic algorithms for hashing. (SHA-512)



Hashing Passwords

- (same key == same hashed value) Issue?
- There are typical user passwords that are common ... the hashed value could be predictable!
- Solution? We add SALT
- SALT: a random value added to the password before it is hashed, making it harder to predict (random bytes).
- The used SALT is private (hidden from outsiders)

Password	Hash
123456	e10adc3949ba59abbe56e057f20f883e
password	5f4dcc3b5aa765d61d8327deb882cf99
12345	827ccb0eea8a706c4c34a16891f84e7b
12345678	25d55ad283aa400af464c76d713c07ad
football	37b4e2d82900d5e94b8da524fbeb33c0
qwerty	d8578edf8458ce06fbc5bb76a58c5ca4
1234567890	e807f1fcf82d132f9bb018ca6738a19f
1234567	fcea920f7412b5da7be0cf42b8c93759
princess	8afa847f50a716e64932d995c8e7435a
1234	81dc9bdb52d04dc20036dbd8313ed055
login	d56b699830e77ba53855679cb1d252da
welcome	40be4e59b9a2a2b5dffb918c0e86b3d7
solo	5653c6b1f51852a6351ec69c8452abc6
abc123	e99a18c428cb38d5f260853678922e03
admin	21232f297a57a5a743894a0e4a801fc3

Deployment

Deployment

- PaaS abstracts the concern of infrastructure.
- PaaS takes care of servers, load-balancers, reverse proxies, or restarting the application on a crash
- Some popular PaaS:





Deployment

- Docker on the other-hand allows having more control over the deployment
- We can create an image of our application and deploy the image to any computer we want.
- You need to have your own servers and maintain the infrastructure yourself.



THANKS!