

# AUTENTICACIÓN

# CONCEPTOS

- Cuentas de usuario
  - Crear modelo User mediante Mongoose
    - Validaciones
    - Métodos específicos
    - ...
- Hashing
- Creación y envío de tokens mediante **json web tokens (jwt)**

# DOCUMENTO DE USUARIO

- Nos servirá para construir el modelo
  - Observa que el password lo hasheamos
- Tokens: un array de objetos
  - Token de acceso (puede haber varios: móvil, pc...)
  - Token de resetear password
  - Token de "uso"

```
{  
  email: 'pepe@pepe.com',  
  password: 'añdfkjasdñfkljad ñ',  
  tokens: [{  
    access: 'auth',  
    token: 'adfadsfñj3ñjkr3'  
  }]  
}
```



# HASHING

- El valor de la contraseña se guarda hasheado
- No se puede deshashear:
  - Se genera un checksum o huella digital
  - Es irreversible
    - No es un sistema de encriptación, porque no se puede desenscriptar
    - Incluso podría pasar (raro) que dos valores diferentes tuvieran el mismo hash
- Utilizaremos librerías específicas

# WORKFLOW PARA EL LOGIN DE USUARIO

- El usuario introduce sus datos (email y contraseña)
- Los datos viajan por la red encriptados (https)
- Proxy inverso, por ej. Apache o Nginx que descripta los datos y los reenvía al Express
- Express lo recibirá en su ruta de usuarios **POST /users** y lo procesa mediante un controlador

- ¿Existe documento en la colección de Users con ese email?
  - Si:
    - Se hashea la contraseña enviada
    - Se compara con el campo password del documento. ¿Es la misma?
      - Si: se le envia token de tipo auth para sucesivas peticiones
      - No: se envía un código http de error (400)
  - No: Se le envía un código http de error (400 o 404)

# MODELO DE USUARIO

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema

var UserSchema = new Schema({
  email: {
    type: String,
    required: true,
    trim: true,
    minlength: 1,
    unique: true
  },
  password: {
    type: String,
    require: true,
    minlength: 6
  }
})
```



# TIPOS DE VALIDACIÓN DE EMAIL

- El email es correcto (expresión regular)
- El email existe: Servicio de verificación (API)
  - Formato de email, dominio e email válido
    - No todos los clientes SMTP dan respuesta
    - Es necesario guardar una base de datos de emails ¡Y este servicio se paga!
      - Similar a lo que pasa con un dns inverso
- El email existe y es propio: activación de cuenta
  - Podemos utilizar algún paquete ya preparado
  - A medida + `nodemailer`(envio de correos)

# VALIDACIONES PERSONALIZADAS

- *Ver custom validators* en la documentación de Mongoose:

```
var userSchema = new Schema({
  phone: {
    type: String,
    validate: {
      validator: function(v) {
        return /\d{3}-\d{3}-\d{4}/.test(v);
      },
      message: props => `${props.value} is not a valid phone n
    },
    required: [true, 'User phone number required']
  }
});
```

# VALIDAR FORMATO EMAIL

- Usaremos el [módulo npm validator](#)

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema
const validator = require('validator')
var UserSchema = new mongoose.Schema({
  email: {
    type: String,
    required: true,
    trim: true,
    minlength: 1,
    unique: true,
    validate: {
      validator: validator.isEmail,
      message: '{VALUE} is not a valid email'
    }
  },
},
```

# CREAR USUARIOS

# PREPARAR LOS TESTS

- Tests para la ruta POST /users
- Esta es la salida que queríamos tener:

Usuarios

POST /users

- ✓ Debería crear un nuevo usuario
- ✓ Debería dar errores de validación si el email es invál
- ✓ Debería dar errores de validación si la contraseña es
- ✓ No debería crear el usuario si ya existe otro con ese

# CARGA DE USUARIOS PARA TESTS

- Fichero utils.js

```
const { ObjectId } = require('mongodb')

const userOneId = new ObjectId()
const userTwoId = new ObjectId()

const User = require('../app/models/User')

const users = [
  {
    _id: userOneId,
    email: 'prueba@prueba.com',
    password: 'password1'
  },
  {
    _id: userTwoId,
```

# TESTS

```
/* global describe it beforeEach */
const request = require('supertest')
const expect = require('chai').expect
const validator = require('validator')

const { loadUsers, users } = require('./utils')

beforeEach(loadUsers)

const User = require('../app/models/User')

/* obtenemos nuestra api rest que vamos a testear */

const app = require('../app/server')
```

# RUTA PARA CREAR USUARIOS

- Creamos fichero *routes/users.js*

```
const router = require('express').Router()
const usersController = require('../controllers/usersController')

router.post('/', (req, res) => {
  usersController.create(req, res)
})

module.exports = router
```



- Anidamos la ruta desde nuestro router principal (/api)

```
const router = require('express').Router()
const cervezasRouter = require('./cervezas')
const usersRouter = require('./users')

router.get('/', (req, res) => {
  res.json({ mensaje: 'Bienvenido a nuestra api' })
})

router.use('/cervezas', cervezasRouter)
router.use('/users', usersRouter)

module.exports = router
```

# CONTROLADOR PARA CREAR USUARIOS

- Implementamos el método create
- Comprobamos que los tests se pongan en verde :-)

```
const User = require('../models/User')

const create = (req, res) => {
  // recogemos los datos del body que nos interesen:
  const { email, password } = req.body
  const user = new User({ email, password })
  user.save((err, user) => {
    if (err) {
      return res.status(400).json({
        message: 'Error al guardar el usuario',
        error: err
      })
    }
    return res.status(201).json(user)
  })
}
```



# **TOKENS: CONCEPTOS GENERALES**

# NECESIDAD

- Las rutas son públicas
- Algunas queremos hacerlas privadas
  - Solo accederán usuarios autenticados
  - Al autenticarse se obtiene un token
  - El token se envía en la petición para acceder a la ruta privada

# HASHING

- Instalación librería

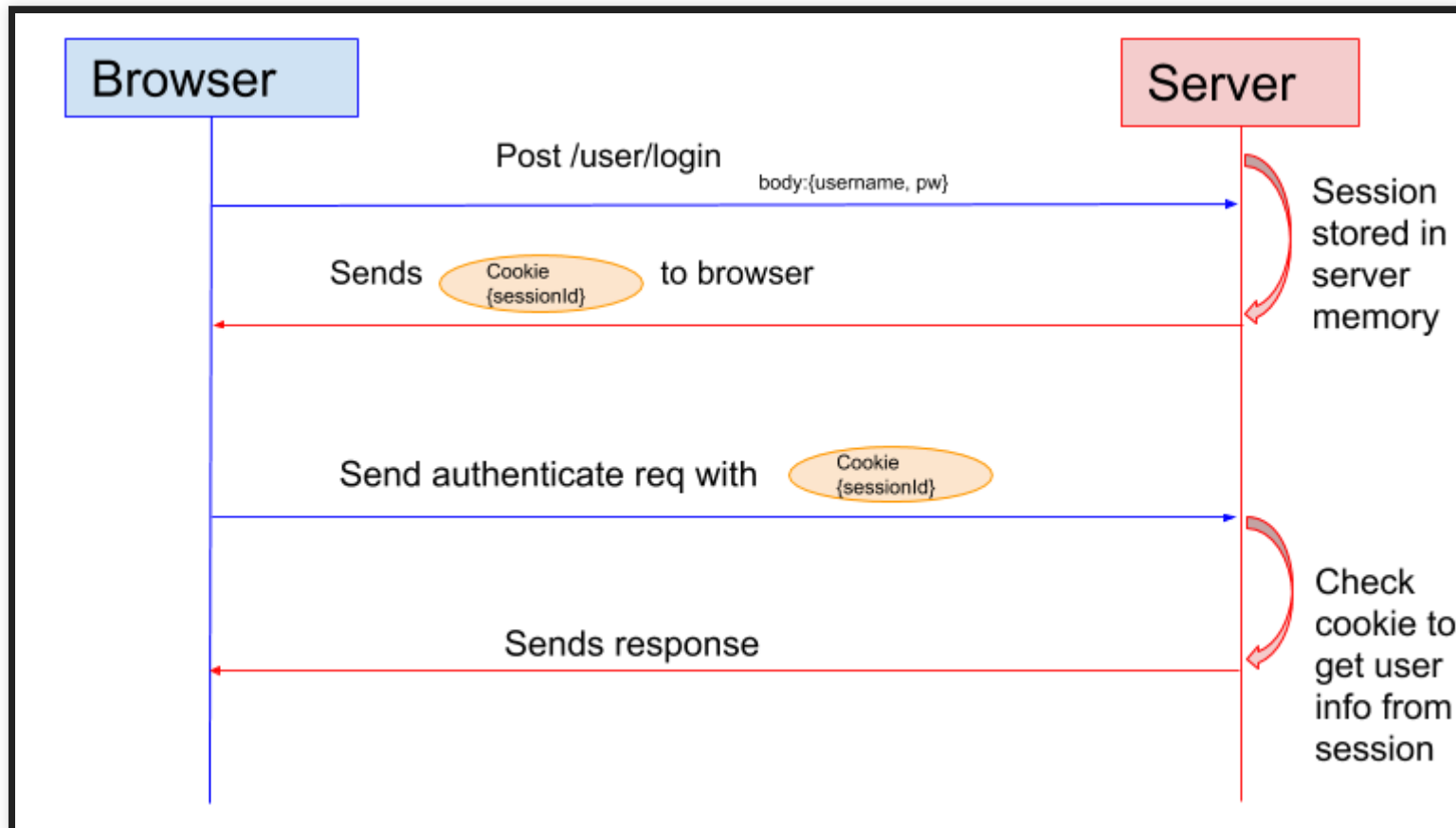
```
npm i -S crypto-js
```

- Uso librería:

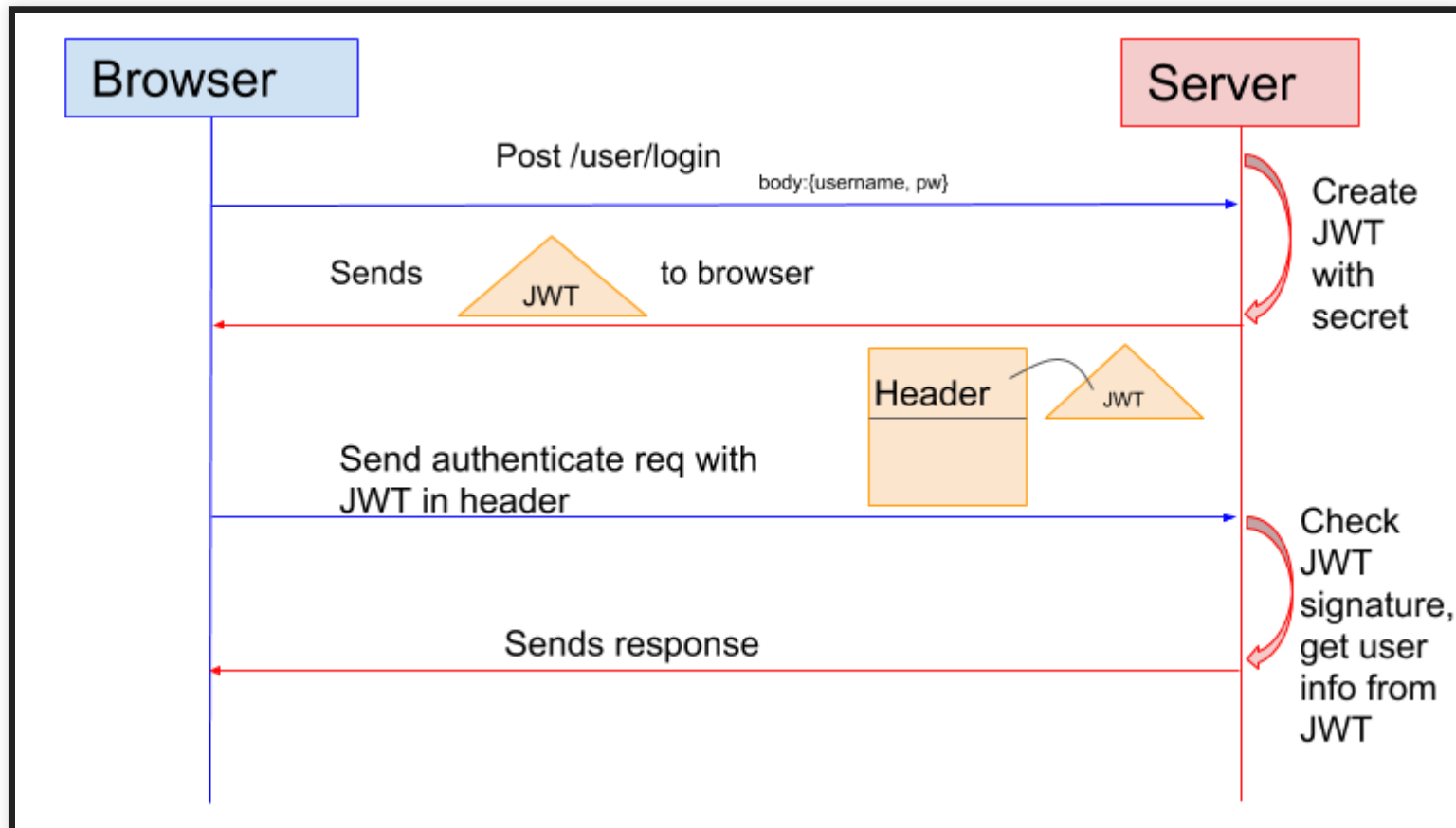
```
const {SHA256} = require('crypto-js')
const password='Esta es mi contraseña'
const hashedPassword = SHA256(password)
console.log(`Password: ${password}`)
console.log(`Hashed Password: ${hashedPassword}`)
```

# TOKENS VS SESIONES

- La conexión http no es persistente
  - Uso de sesiones
    - Se guarda el SESSION\_ID en cookies
    - Las cookies se mandan de forma automática
  - Uso de tokens
    - Se guardan en *local storage*: envío manual
    - Se guardan en cookies:
      - Envío automático
      - Problemas en apis multidominio
    - Mayor escalabilidad







- Cada vez que un usuario hace peticiones si está autenticado debe enviar sus datos
  - id de usuario
  - permisos
  - ....

```
var data = {id: 4}
```

- Los datos no se deben modificar (suplantación de identidad)
- Debemos enviar otra cosa... algo que nos de integridad:

```
var token = {  
  data,  
  hash: SHA256(JSON.stringify(data)).toString()  
}
```

- El usuario podría cambiar también el hash
  - Necesitamos "sal en el hash"
  - Añadir algún valor aleatorio que solo conozca el generador del hash

```
var token = {  
  data,  
  hash: SHA256(JSON.stringify(data) + 'somesecret').toString()  
}
```

# COMPROBACIÓN TOKEN

- Nos llega el siguiente token del usuario:

```
var token = {  
  data: {id: 5},  
  hash: "añdalñdkfñadj añsdlfgjk añdlk añlfdj dfñjk jjafdkjdfd"  
}
```

- Comprobamos si es correcto:

```
var resultHash = SHA256(JSON.stringify(token.data) + 'somesecr'  
if (resultHash === token.hash) {  
  console.log('Los datos no se han modificado')  
} else {  
  console.log('Los datos están modificados, no se puede conf'  
}
```

# MANIPULACIÓN DEL TOKEN

- Generamos el token desde el cliente (no conocemos la sal):

```
var token = {  
  data: {id: 5},  
  hash: SHA256(JSON.stringify(token.data))  
}
```

- El token no coincidirá porque se desconoce la sal.

```
var resultHash = SHA256(JSON.stringify(token.data) + 'somesecr  
if (resultHash === token.hash) {  
  console.log('Los datos no se han modificado')  
} else {  
  console.log('Los datos están modificados, no se puede conf  
}
```

# ESTANDAR JWT

- Token compuesto de:
  - **Header:** algoritmo y tipo de token (jwt en nuestro caso)
  - **Payload:** los datos
  - **Verify Signature:** hash para comprobar integridad
- Existen librerías que lo hacen sencillo
- Se pueden [generar y comprobar online](#)

# TOKEN: IMPLEMENTACIÓN



# LIBRERÍA JSONWEBTOKEN

- Utilizaremos `jsonwebtoken`
- Realizada por la `empresa Auth0`
- Tiene basicamente dos funciones
  - `jwt.sign`: para firmar el token
  - `jwt.verify`: para verificar el token

# EJEMPLO DE USO

- Si el token ha sido manipulado arrojará un error de **Invalid Signature**.
- Se pueden [comprobar online](#)

```
const jwt = require('jsonwebtoken')
var data = {id: 1}
var token = jwt.sign(data, 'privatePassword')
console.log(token)
var decoded = jwt.verify(token, 'privatePassword')
console.log(decoded)
```

# GENERAR TOKEN

- Lo haremos en el modelo, al crear el usuario.
  - Seguimos patrón MVC (thin controller, fat model)
  - Reutilización de código
- Definimos un método a nivel de instancia, ¡sin arrow functions! (no acceden a *this*)

```
UserSchema.methods.methodName = function () {....}
```

- Si definimos el método a nivel de modelo no podríamos usar *this* (datos de la instancia)

```
UserSchema.statics.methodName = function () {....}
```

```
const jwt = require('jsonwebtoken')

UserSchema.methods.generateAuthToken = function () {
  var user = this
  var access = 'auth'
  var token = jwt.sign({_id: user._id.toHexString(), access},
    user.tokens.push({access, token})
  return user.save().then(() => {
    return token
  })
}
```

# TEST CON ENVIO DE TOKEN

- En el caso de que el POST /user sea correcto hay que comprobar que se envía el token
  - El token lo envío como header propio (*x-auth*)

```
it('Debería crear un nuevo usuario', done => {
  const email = 'curso@curso.com'
  const password = 'P@ssw0rd'

  request(app)
    .post('/api/users')
    .send({ email, password })
    .expect(201)
    .expect(res => {
      expect(res.headers['x-auth']).to.exist
      expect(res.body._id).to.exist
      expect(res.body.email).to.satisfy(validator.isEmail)
    })
    .end(err => {
      if (err) {
```



# IMPLEMENTACIÓN DEL ENVIO DE TOKEN

- En el método `create` del `UserController`
- ¡Ojo, varias llamadas asíncronas!
  - Método `save()`
  - Método `generateAuthToken()`
- Reescribiremos el código para utilizar promesas
  - Evitaremos una anidación excesiva
- Comprobamos que se cumplen los tests

```
const User = require('../models/User')

const create = (req, res) => {
  const { email, password } = req.body
  const user = new User({ email, password })
  user
    .save()
    .then(() => {
      return user.generateAuthToken()
    })
    .then(token => {
      res.header('x-auth', token).status(201).send(user)
    })
    .catch(e => {
      res.status(400).send(e)
    })
}
```



# RUTAS PRIVADAS

# WORKFLOW DE ACCESO A RUTAS PRIVADAS

- Se solicita un Auth token
- Se valida el token
- Se busca el usuario asociado al token
- Y entonces se accede a la ruta privada

# CONVERTIR RUTA PÚBLICA A PRIVADA

- La ruta de *signUp* o *signIn* siempre debe ser pública
- Puede haber muchas rutas privadas
  - Usaremos un middleware en dichas rutas para no duplicar código
- Creemos un ejemplo de ruta privada, solicitando el perfil del usuario:

```
app.get('/users/me', (req, res)=>{
  var token = req.header('x-auth')
  User.findByToken(token).then((user)=>{
    if (!user) {

    }
    res.send(user)
  })
})
```

# MÉTODO *FINDBYTOKEN*

- Se define en el modelo
  - Es aquí donde implementamos el `verify` y buscamos documento en la colección `Users`
- También podría ser todo en el middleware si no accedemos a bbdd
  - Este método podría ser sin base de datos, un simple *jwt.verify*

```
UserSchema.statics.findByToken = function (token) {  
  const User = this  
  let decoded  
  
  try {  
    decoded = jwt.verify(token, 'abc123')  
  } catch (e) {  
    return Promise.reject(e)  
  }  
  
  return User.findOne({  
    _id: decoded._id,  
    'tokens.token': token,  
    'tokens.access': 'auth'  
  })  
}
```

# MIDDLEWARE DE AUTENTICACIÓN

- Comprueba si el token es correcto
  - Llama al método findByToken del modelo User
- Si el token es correcto lo añade al request
- Si el token es incorrecto, termina la comunicación

```
const { User } = require('../models/user')

const authenticate = (req, res, next) => {
  const token = req.header('x-auth')

  User.findByToken(token).then((user) => {
    if (!user) {
      return Promise.reject()
    }
    req.user = user
    req.token = token
    next()
  }).catch((e) => {
    res.status(401).send()
  })
})
```



# USAR MIDDLEWARE AUTHENTICATE

- Cargamos el middleware authenticate
  - Lo insertamos como segundo parámetro en las rutas privadas

```
const authenticate = require('./middleware/authenticate')
app.get('/users/me', authenticate, (req, res)=>{
  res.send(req.user)
})
})
```

# CONTRASEÑA DEL USUARIO

# HASH DE LA CONTRASEÑA DE USUARIO

- Ahora el usuario hace un POST /users
  - Su contraseña se guarda como texto plano
- ¿Qué queremos?
  - Guardaremos la contraseña encriptada
  - Utilizaremos una sal específica para cada encriptación

# INSTALACIÓN Y USO DE BCRYPTJS

- Hay otras librerías basadas en bcrypt, por ej. bcrypt.
- La que usamos esta toda hecha en js y es más portable, da menos problemas

```
npm i -S bcryptjs
```

- Se deben ejecutar dos métodos
  - **bcrypt.genSalt** para la generación de la sal
  - **bcrypt.hash** para crear el hash

# EJEMPLO DE USO DE BCRIPTJS

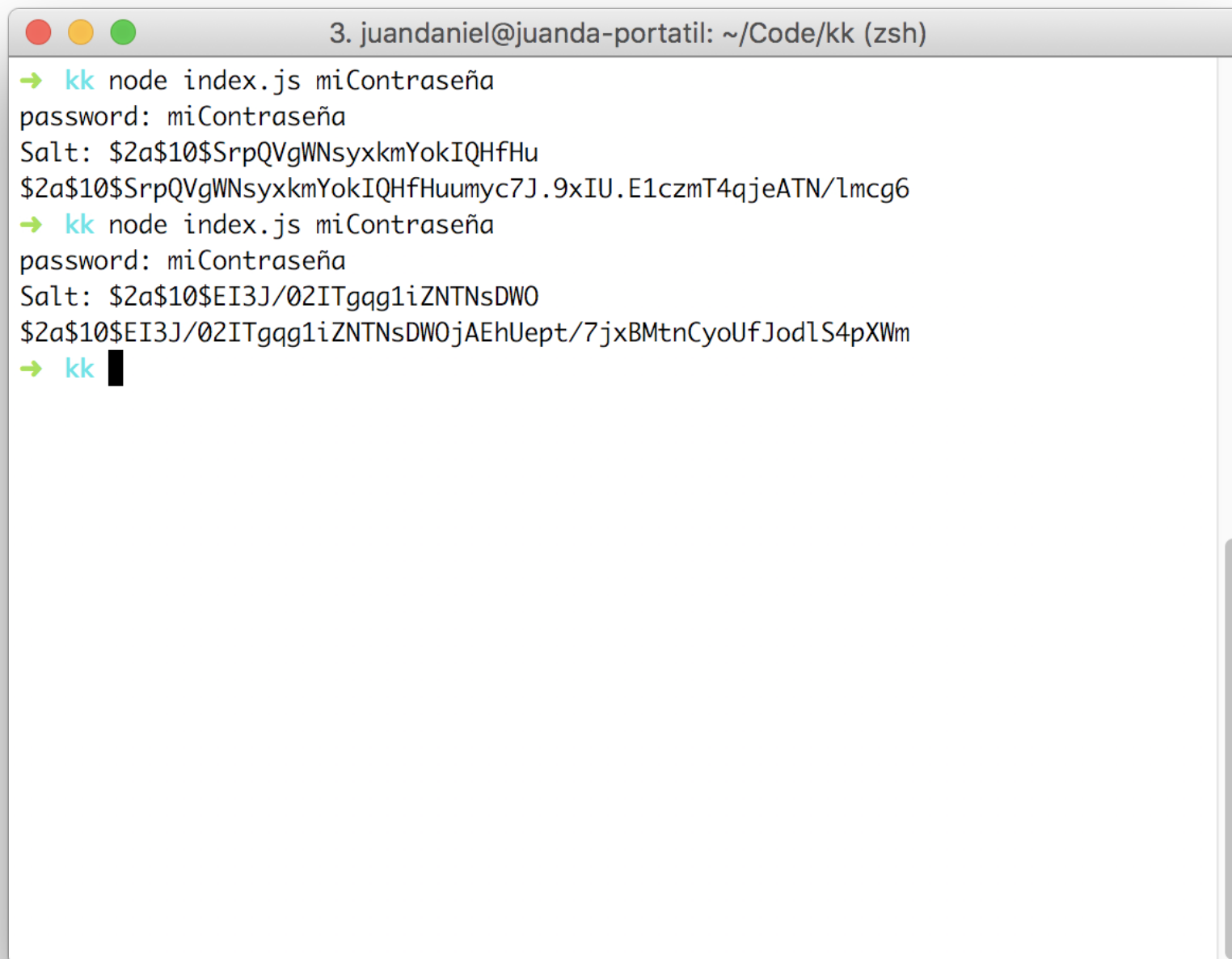
```
const bcrypt = require('bcryptjs');
const password = process.argv[2];
console.log(`password: ${password} `);
bcrypt.genSalt(10, (err, salt) => {
  console.log(`Salt: ${salt}`)
  bcrypt.hash(password, salt, (err, hash) => {
    console.log(hash) // esto es lo que queremos guardar en
  })
})
```

# SALIDA DE BCrypt

- Guardamos la sal con la contraseña

```
$<algorithm>$<iterations>$<salt>$<hash>
```

- el algoritmo de hash
- El número de iteraciones o factor de trabajo
- La sal aleatoria
- La contraseña resultante o hash



```
3. juandaniel@juanda-portatil: ~/Code/kk (zsh)
→ kk node index.js miContraseña
password: miContraseña
Salt: $2a$10$SrpQVgWNsyxkmYokIQHfHu
$2a$10$SrpQVgWNsyxkmYokIQHfHuumyc7J.9xIU.E1czmT4qjeATN/lmcg6
→ kk node index.js miContraseña
password: miContraseña
Salt: $2a$10$EI3J/02ITgqg1iZNTNsDWO
$2a$10$EI3J/02ITgqg1iZNTNsDWOjAEhUept/7jxBMtnCyoUfJodlS4pXWm
→ kk
```





# CHEQUEO DE CONTRASEÑA

```
bcrypt.compare(password, hashedPassword, (err, res)=>{  
  // res es true o false  
})
```

# EJERCICIO BCrypt

- Utiliza `bcrypt.compara`:
  - Comprueba que devuelve `true` si el password es correcto
  - Comprueba que devuelve `false` si el password es erróneo

# SOLUCIÓN EJECICIO BCRIPT

```
const bcrypt = require('bcryptjs');
const password = process.argv[2];

hashedPassword='$2a$10$dH5q2dYWbwLMXrPnCMQ52epdQlyvhUqZrUg5iEh
bcrypt.compare(password, hashedPassword, (err, res)=>{
  if (err) console.log(`Error: ${err}`)
  console.log(`El resultado de la comparación es: ${res}`)
})
```

# MONGOOSE MIDDLEWARE

- Permite ejecutar cierto código antes o después de ciertos eventos
  - Antes de guardar el documento (usuario) cambiaremos el password por el hash
- [Ver documentación](#)

```
var schema = new Schema(..);
schema.pre('save', function(next) {
  // do stuff
  next();
});
```

# EJERCICIO IMPLEMENTAR MIDDLEWARE

- Al guardar un usuario, su contraseña se debe cambiar por el hash
- Si la contraseña no ha cambiado, no se debe modificar el hash
  - El método save puede llamarse en una actualización por ej.
  - Ayúdate del método isModified que proporciona MongoDB.

# ESQUEMA DE AYUDA

```
UserSchema.pre('save', function (next) {  
  var user = this  
  
  if (user.isModified('password')) {  
    // user.password  
  
    // user.password = hash  
    // next()  
  } else {  
    next()  
  }  
})
```

# SOLUCIÓN IMPLEMENTACIÓN MIDDLEWARE

```
UserSchema.pre('save', function (next) {  
  var user = this  
  
  if (user.isModified('password')) {  
    bcrypt.genSalt(10, (err, salt) => {  
      bcrypt.hash(user.password, salt, (err, hash) => {  
        user.password = hash  
        next()  
      })  
    })  
  }  
  else {  
    next()  
  }  
})
```

# TEST DE FUNCIONAMIENTO

```
it('Debería crear un nuevo usuario', done => {  
  const email = 'curso@curso.com'  
  const password = 'P@ssw0rd'  
  
  request(app)  
    .post('/api/users')  
    .send({ email, password })  
    .expect(201)  
    .expect(res => {  
      expect(res.headers['x-auth']).to.exist  
      expect(res.body._id).to.exist  
      expect(res.body.email).to.satisfy(validator.isEmail)  
    })  
    .end(err => {  
      if (err) {
```