Understanding the code.

The User model

```
import bcrypt from 'bcrypt';
import config from 'config';
import mongoose from 'mongoose';
import { User } from '../core/models';

export interface userDocument extends User, mongoose.Document {
  createdAt: Date;
  updatedAt: Date;
  comparePassword(userPassword: string): Promise<Boolean>;
}
// instead of extending user, we could do this
export interface userDocument extends mongoose.Document {
  email: string;
  name: string;
  password: string;
  createdAt: Date;
  updatedAt: Date;
  comparePassword(userPassword: string): Promise<Boolean>;
}

const userSchema = new mongoose.Schema(
  {
    email: { type: String, required: true, unique: true },
    name: { type: String, required: true },
    password: { type: String, required: false },
  },
  {
    timestamps: true,
  }
);


userSchema.pre("save", async function (next: any) {
  let user = this as userDocument;

  if (!user.isModified("password")) {
    return next();
  }

  const salt = await bcrypt.genSalt(config.get<number>("saltWorkFactor"));
  const hash = await bcrypt.hashSync(user.password!, salt);
```

```
  user.password = hash;

  return next();
});

userSchema.methods.comparePassword = async function (inputPassword: string): Promise<Boolean> {
  let user = this as userDocument;

  return bcrypt.compare(inputPassword, user.password!).catch((e:any) => false);
};

export const userModel = mongoose.model<userDocument>("User", userSchema);
```

A Mongoose schema is a blueprint for how a MongoDB document should look like. It defines the structure and data types for the fields in a MongoDB document. The schema also defines any required fields, default values, and any custom methods associated with the document.

For example, let's say you're building a social media application, and you need to store information about the users of the platform. A Mongoose schema for a user document could look like this:

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  age: { type: Number, required: false },
  bio: { type: String, required: false }
});
```

Once the schema is defined, you can use it to create a Mongoose model, which is a class that provides a convenient API for working with the user data in the MongoDB database. You can use the model to create new user documents, read existing documents, update documents, and delete documents.

```
export interface userDocument extends User, mongoose.Document {
  createdAt: Date;
  updatedAt: Date;
  comparePassword(userPassword: string): Promise<Boolean>;
}
```

▼ The code defines an "interface" named "`userDocument`" in TypeScript, which is a blueprint for objects that specify the structure and type of properties and methods an object must have. This is similar to Java interfaces in that it defines a contract that objects must follow, but in TypeScript interfaces can include property types and method signatures.

]The "`userDocument`" interface extends two other interfaces/types: "`User`" and "`mongoose.Document`". “User" is not defined in this code snippet, so it can be assumed it's a custom interface/type that represents the structure and properties of a user. "`mongoose.Document`" is a type provided by the Mongoose library, which is a popular ODM (Object Document Mapper) for MongoDB in Node.js. It represents a MongoDB document and provides additional properties and methods specific to MongoDB documents, such as "`createdAt"` and "`updatedAt`".

By using "extends", "`userDocument`" interface is combining the properties and methods from both "User" and "`mongoose.Document`". It then adds two additional properties "`createdAt`" and "`updatedAt`" of type "Date", and a method named "`comparePassword`" that takes a "`userPassword`" string as input and returns a "Promise<Boolean>" indicating the result of the password comparison.

In this case, using an interface is appropriate because it allows the developer to define a custom type that can be used throughout their code to ensure that objects conform to a certain structure and behavior. It could not have been a class because a class must have a constructor and instance methods, whereas an interface only defines a blueprint of properties and methods.

In conclusion, "`userDocument`" is a custom TypeScript interface that represents a User document stored in MongoDB and includes properties and methods from both "User" and "`mongoose.Document`", as well as additional properties and methods specific to the user document. "`mongoose.Document`" provides additional properties and methods specific to MongoDB documents and is used to interact with MongoDB database.

In summary, the **`userSchema`**
defines the structure and validation rules for a MongoDB document, while the
`userDocument`
extends the properties and methods of a Mongoose Document to add custom properties and methods that are specific to a user document.

```
userSchema.pre("save", async function (next: any) {
  let user = this as userDocument;

  if (!user.isModified("password")) {
    return next();
  }

  const salt = await bcrypt.genSalt(config.get<number>("saltWorkFactor"));
  const hash = await bcrypt.hashSync(user.password!, salt);

  user.password = hash;

  return next();
});
```

The code you provided is a Mongoose middleware function that is defined using the `pre` method on the `userSchema`.

In Mongoose, middleware functions are hooks that run before or after a certain event, such as saving a document or deleting a document. The `pre` method is used to define a middleware function that runs before the specified event, in this case, the "save" event.

The middleware function takes two arguments, `next` and `this`, where `next` is a callback function that must be called to move to the next middleware in the chain, and `this` refers to the document that the middleware is being run for.

The function casts `this` to `userDocument` using the `as` operator, which allows the function to access the custom properties and methods of the `userDocument` interface.

The function checks if the password field has been modified using the `isModified` method. If the password field has not been modified, the function calls `next` and returns, allowing the save operation to proceed without modifying the password.

If the password field has been modified, the function generates a salt using the `bcrypt.genSalt` method, where `bcrypt` is a library for hashing passwords. The salt is generated with a number of rounds specified in the `saltWorkFactor` configuration value.

The function then hashes the password using the `bcrypt.hashSync` method and the generated salt. The hash is stored in the `hash` variable, and the original password is replaced with the hash.

Finally, the function calls `next` and returns, allowing the save operation to proceed.

In summary, the middleware function you provided performs password hashing before saving a user document. It runs before the "save" event and checks if the password field

has been modified. If it has been modified, the function generates a salt and hashes the password, replacing the original password with the hash.

```
export const userModel = mongoose.model<userDocument>("User", userSchema);
```

The code you provided creates a Mongoose model using the `mongoose.model` method. The model is created based on the `userSchema` and named "User".

The purpose of creating a Mongoose model is to define a blueprint for documents in a MongoDB collection. By creating a model, you can perform various operations on the documents in the collection, such as querying, updating, or deleting documents. The model acts as a wrapper around the collection and provides a convenient API for working with the documents.

The first argument of the `mongoose.model` method is the name of the model, which will be used to create the name of the collection in MongoDB. The second argument is the schema that the model should use, which in this case is the `userSchema` defined earlier.

The model is declared as an instance of `mongoose.Model<userDocument>`, which means it is a generic model that works with documents of type `userDocument`. By specifying the `userDocument` type, the model can take advantage of the custom properties and methods defined in the `userDocument` interface.

In summary, the line of code you provided creates a Mongoose model based on the `userSchema` and named "User". The model provides a convenient API for working with the documents in the collection and takes advantage of the custom properties and methods defined in the `userDocument` interface.

The User Service

```
import { User } from '../core/models';
import { userModel } from './../models/user.model';

export async function createUser(input:User) {
    try{
        return await userModel.create(input);
    }
    catch(e:any){

        throw new Error(e);
```

```
    }
}
```

The function `userModel.create(input)` creates a new user document and saves it to the database collection represented by the `userModel`. This function is a method provided by Mongoose, which is an ODM (Object Document Mapping) library for MongoDB and Node.js.

Step by step:

1. `userModel` is the Mongoose model for the User collection in the database.
2. The `create` method is called on the `userModel` and is passed the `input` argument, which is an object of the `User` type.
3. The `create` method validates the input and creates a new user document based on the input.
4. The `create` method saves the new user document to the corresponding User collection in the database.
5. The method returns a Promise that resolves to the newly created user document.

The User Controller

```
import { createUser } from '../services/user.service';
import { HttpStatusCode } from './../core/enums/http-status-codes.enum';
import { log } from '../utils';
import { Request, Response } from 'express';

export async function createUserHandler(req:Request , res:Response) {
    try{
        const user = await createUser(req.body);
        //return res.status(HttpStatusCode.OK).send(user);
    }
    catch(e:any){
        log.error(e);
        return res.status(HttpStatusCode.CONFLICT).send(e.message);
    }
}
```

The `createUserHandler` function is a Express.js HTTP endpoint handler, which is triggered when a user makes a request to the specified route (Api/users).

It takes two arguments, `req` and `res`, representing the HTTP request and response objects respectively.

The function invokes the `createUser` function and passes in the request body `req.body`.

The purpose of this function is to handle the incoming request, invoke the `createUser` function and return the result back to the client via the response object `res`.

If this function is not implemented, then there will not be any code to handle the incoming request and send back the response, leading to a failed request and error.

The User Router

```
function userRouters(app: Express) {
    app.post("/api/users",createUserHandler);
}
```

The `userRouters` function sets up a route in an ExpressJS application. It takes an `app` argument, which is an instance of the ExpressJS `Express` class.

The code sets up a `POST` endpoint at the route `/api/users`. When a `POST` request is made to this endpoint, the `createUserHandler` function will be invoked as a callback.

The `createUserHandler` is an Express.js route handler that is called when a user makes a POST request to the "/api/users" endpoint. The purpose of this function is to handle the incoming request, process it, and return an appropriate response.

When a user makes a POST request to "/api/users", the `createUserHandler` function is executed. Inside the function, it calls the `createUser` function, passing in the request body as an argument. This is the data that the user sent in the request, and it will be used to create a new user in the database.

The `createUser` function is responsible for creating the user in the database and returning the newly created user.

The `createRouters` function sets up a new Express.js route, attaching the `createUserHandler` function as the handler for POST requests to the "/api/users" endpoint.

In general, the router is responsible for mapping URLs to the code that handles the request. When a user makes a request to an endpoint, the router maps the URL to a specific route handler, which then handles the request and returns the appropriate response.

When a POST request to the `/api/users` endpoint is made, the following sequence of actions occur:

1. The userRouter function is triggered, and it listens for a POST request to the `/api/users` endpoint.
2. The middleware `validateResource(createUserSchema)` is called. This middleware validates the incoming request data against the `createUserSchema` defined in the `user.schema.ts` file.
3. If the request data is not valid, the `validateResource` middleware will return an error indicating the invalid data.
4. If the request data is valid, the next middleware, `createUserHandler` is called.
5. `createUserHandler` function is triggered. It extracts the request body and passes it to the `createUser` function as an argument.
6. The `createUser` function is called and it attempts to create a new user by calling `userModel.create(input)`, where `input` is the extracted request body.
7. If the user creation is successful, the created user is returned by the function.
8. If the user creation fails, an error is thrown and the catch block in the `createUser` function is executed.
9. `createUserHandler` returns the response to the client, indicating the success or failure of the user creation.