**Supply Chain Simulator**

**Guía del desarrollador**

# Introducción

En esta guía vas a encontrar la información necesaria para poder desarrollar tu propia versión de **Supply Chain Simulator** (**SCS**), todos los datos relevantes sobre las clases y la lógica del sistema. Para esto vas a tener que contar con unos requisitos técnicos básicos para poder entender y aprovechar de mejor manera este framework de simulación.

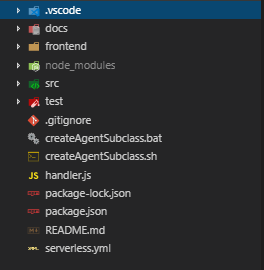
# Requisitos técnicos

Para desarrollar y extender este framework debes de contar con los siguientes conocimientos técnicos:

* Javascript.
* NodeJs.
* HTML.
* CSS.

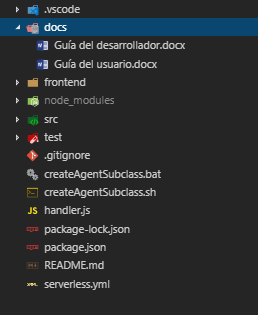
# Estructura de directorios

Esta es la estructura de directorios básica del proyecto y la vamos a ir profundizando a lo largo de toda la guía.



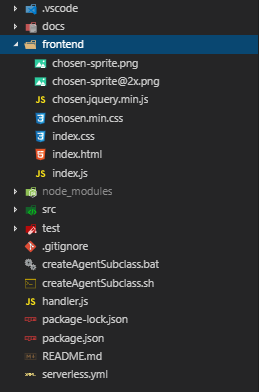
La primer carpeta .vscode la vamos a obviar debido a que es una carpeta de manejo interno del Visual Studio Code, editor de código de Microsoft.

## docs



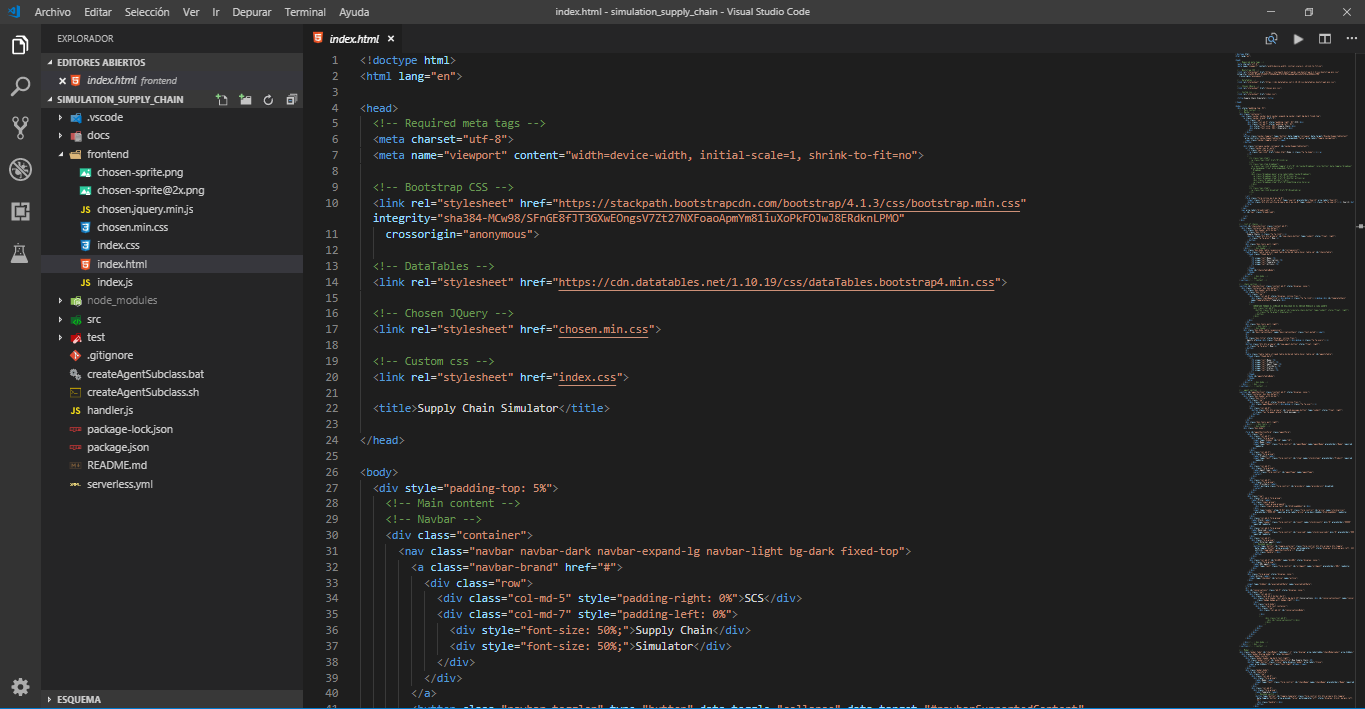
Esta carpeta contiene las Guías del desarrollador y del usuario, material que presenta toda la información necesaria para modificar y hacer uso de este framework de simulación de cadenas de suministros **SCS**.

## frontend



Este directorio contiene los archivos que integran el frontend del sistema. En los archivos index.html, index.js e index.css se encuentra el código necesario para el manejo lógico y visual de esta parte del proyecto y los archivos chosen-sprite.png, chosen-sprite@2x.png, chosen.jquery.min.js y chosen.min.js pertenecen al plugin JQuery, Chosen, por esto es aconsejable dejar intactos los mismos para no correr riesgo de afectar al funcionamiento del plugin.

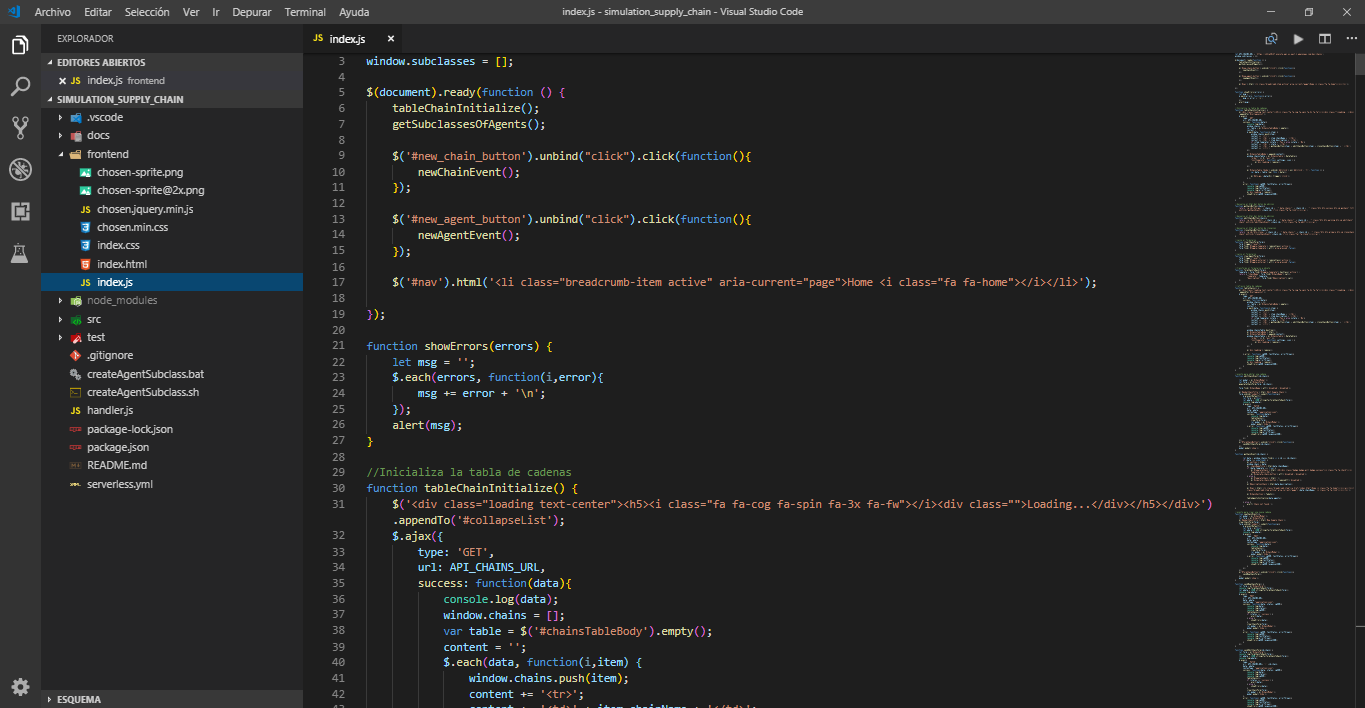
### index.html



Este archivo es el encargado de contener el código HTML del frontend del sistema. El mismo utiliza las librerias:

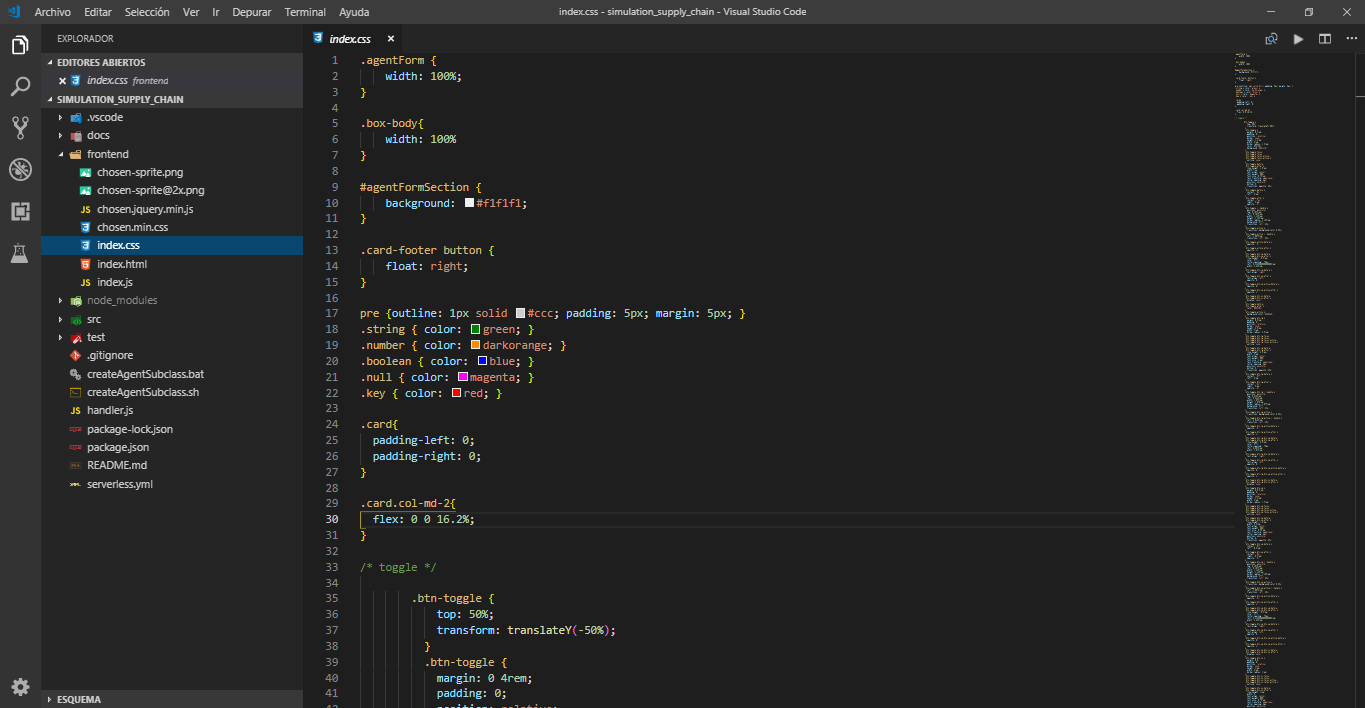
* Bootstrap 4.1.3.
* DataTables 1.10.19.
* Chosen 1.8.7.
* JQuery 3.3.1.
* Popper 1.14.3.
* Fontawesome 5.2.

### index.js



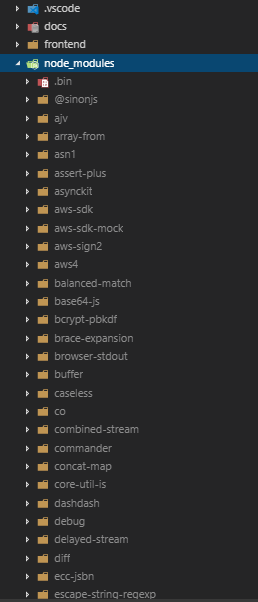
Código JavaScript (JS) que acompaña al presente en index.html para manejar los eventos JavaScript de los elementos del frontend, entre ellos el comportamiento de los botones, modales, llamados Ajax a las funciones lambda situadas en Amazon, entre otras.

### index.css



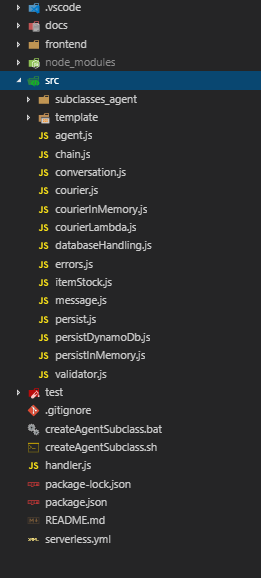
En este archivo se contienen los estilos CSS del frontend del sistema, estilos como por ejemplo los de los elementos toggle, el formato de los tipos de mensajes de las conversaciones, formularios, etc.

## node\_modules



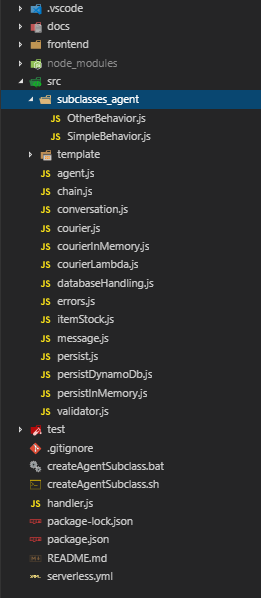
Este directorio contiene los módulos de NodeJs utilizados en el sistema. Es importante no modificar ninguno de estos archivos para no afectar el funcionamiento de los módulos.

## src



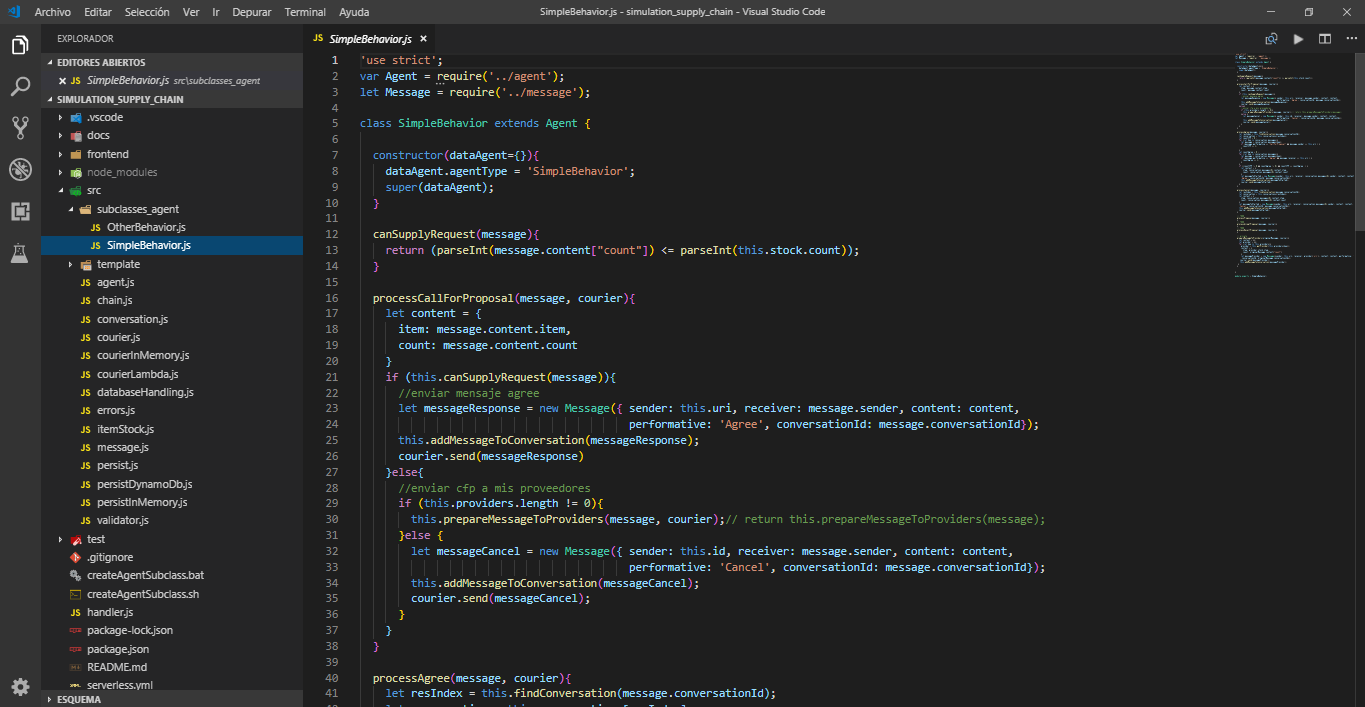
En dicha carpeta se encuentran los archivos JS encargados de manejar la lógica de las clases presentes en las funciones lambda, integrando así el backend de este proyecto. Además, contiene 2 directorios internos detallados a continuación.

### subclases\_agent



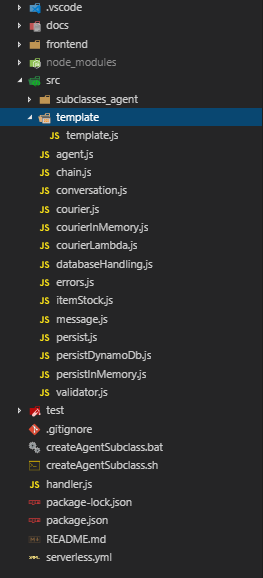
En esta carpeta se contienen las clases JS que heredan su comportamiento de la clase **Agent**, por ejemplo, las clases **OtherBeahvior** o **SimpleBehavior**. Tu puedes definir tus propias subclases de **Agent**, esto se explicará con más detalle en el apartado **“Crea tus propios agentes”**.

#### SimpleBehavior.js



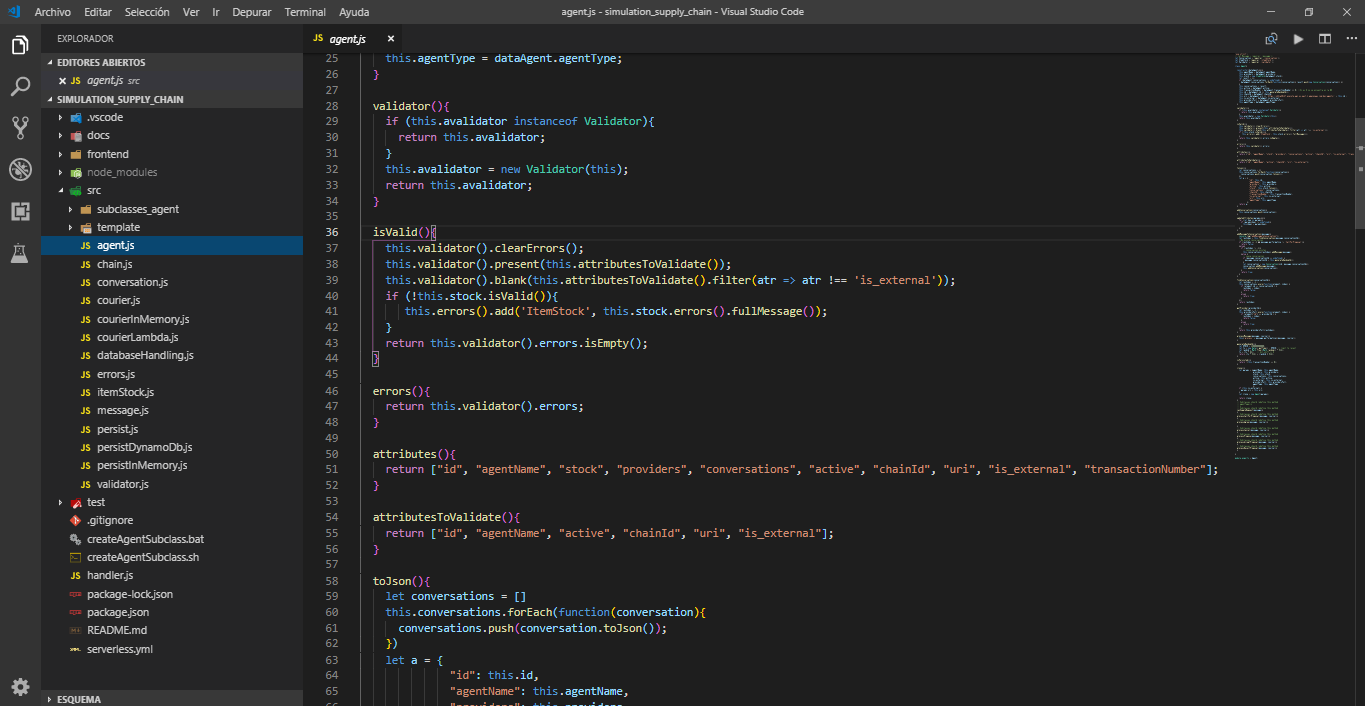
Es una de las subclases de **Agent**, en ella se define el comportamiento que debe tener un agente para por ejemplo procesar un mensaje enviado por otro agente de la cadena de suministros, agregar un nuevo mensaje a una de sus conversaciones, etc.

### template



En el directorio template se encuentra el archivo template.js, que como su nombre lo indica, se utiliza como template de la clase **Agent**, este archivo define a la clase **Template**, clase abstracta que contiene la interfaz necesaria que debe implementar cualquier subclase de **Agent**. Es importante no alterar esta carpeta ni su contenido para no afectar la funcionalidad de crear subclases de **Agent** de forma automática.

### agent.js



Este archivo contiene a la clase **Agent** y toda su lógica. De esta heredan y extienden su comportamiento las clases presentes en el directorio **subclases\_agent** anteriormente descripto.

#### Atributos de la clase **Agent**

Esta clase presenta los siguientes atributos de clase:

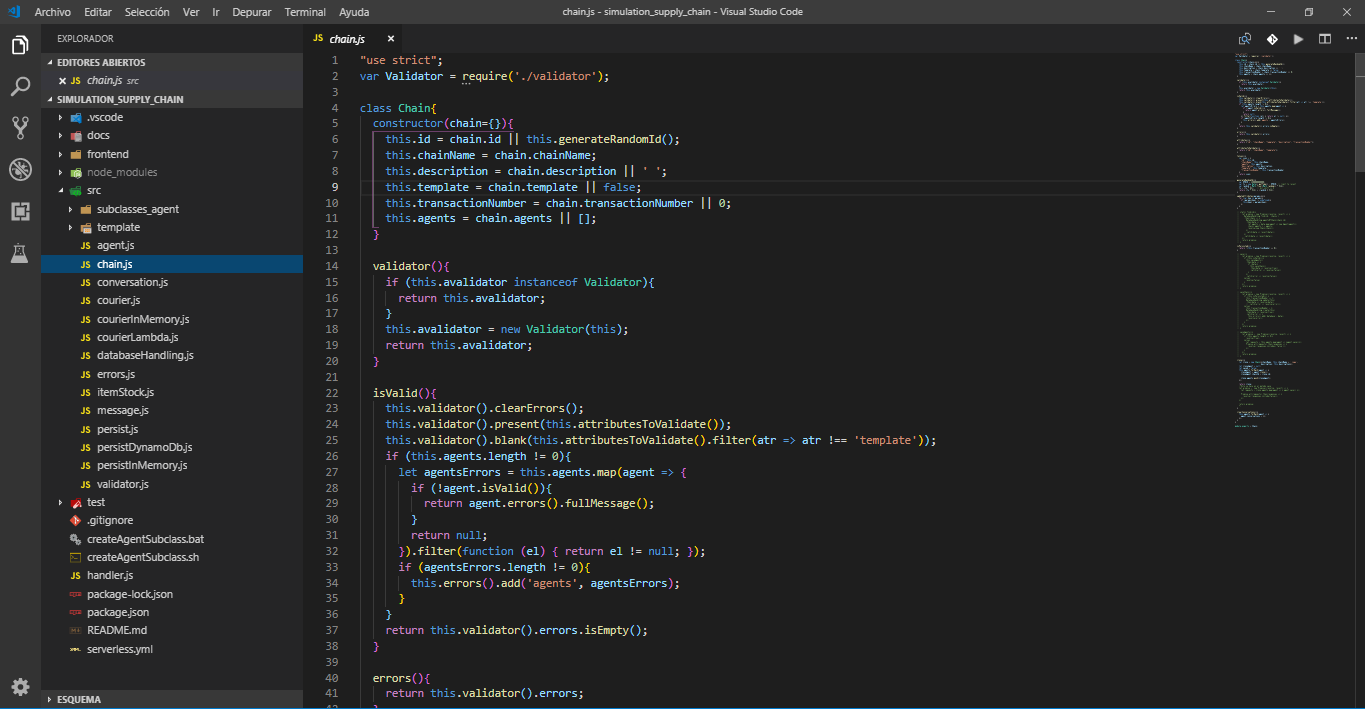
* agentName: (string) representa el nombre del agente.
* providers: (Array<int>) representa el conjunto de identificadores de proveedores del agente.
* stock: (**ItemStock**) representa el stock del agente.
* conversations: (Array<**Conversation**>) representa el conjunto de conversaciones del agente.
* active: (boolean) indica si el agente está activo o no.
* transactionNumber: (int) indica si el agente está en condiciones de ser persistido en la BD o no.
* id: (int) representa el identificador único del agente.
* chainId: (int) representa al identificador único de la cadena a la que pertenece el agente.
* uri: (string) representa la ruta de acceso al agente.
* is\_external: (boolean) indica si el agente es externo o no.
* providersFull: (Array<**Agent**>) representa el conjunto de proveedores del agente.
* agentType: (string) representa la clase del agente.

Todos estos atributos no deben ser modificados para no afectar al comportamiento básico de los agentes del sistema, es por esto que se encuentran en la superclase **Agent**, siendo así heredados a todas sus subclases.

#### Métodos de la clase **Agent**

* constructor: recibe un hash con los atributos básicos de un agente y los instancia.
* validator: utiliza el patrón **Singleton** para instancia un objeto del tipo **Validator.**
* isValid: utilizando el objeto validador instanciado por la función anterior, indica si el agente es válido o no.
* errors: utilizando el objeto validador antes descripto, retorna los errores encontrados por el mismo.
* attributes: función que retorna un arreglo con las variables de clase de la clase **Agent**.
* attributesToValidate: a diferencia de la función anterior, esta devuelve un arreglo con las variables que se validarán en la clase **Agent**.
* toJson: función que retorna un hash con las variables de clase del agente que serán enviadas en una petición HTTP o hacia la base de datos.
* addConversation: este método recibe una conversación por parámetro y la agrega a su colección de conversaciones.
* updateAttributes: dicho método recibe un hash con variables y sus valores y actualiza las variables de clase del agente.
* addMessageToConversation: esta función recibe un mensaje por parámetro y evalúa si pertenece a una de sus conversaciones. Si no pertenece a ninguna de sus conversaciones y además no es del tipo **CallForProposal**, la función retorna **false**. Por el contrario, si el mensaje pertenece a una de sus conversaciones, entonces lo agrega a la misma. Y si este mensaje no pertenece a ninguna de sus conversaciones, pero es del tipo **CallForProposal**, esto quiere decir que es un mensaje nuevo para el agente, por lo tanto, le asigna un nuevo ID de conversación, crea una nueva conversación con ese ID y agrega el mensaje a esta conversación recién creada.
* findConversation: la función recibe un identificador de conversación y busca dentro de su colección de conversaciones si ese ID pertenece a alguna de ellas. Si este ID no coincide con ninguna conversación, la función retorna -1, sino retorna el índice de la conversación dentro del arreglo de conversaciones del agente.
* getProvider: esta función recibe por parámetro un identificador de proveedor y busca dentro de su colección de proveedores si hay algún proveedor que contenga dicho ID. Si ningún proveedor posee este identificador, la función retorna -1, caso contrario retorna al proveedor.
* processMessage: este método recibe por parámetro un mensaje y una instancia de alguna subclase de **Courier**. Dependiendo del tipo del mensaje, este procedimiento ejecuta el método adecuado para procesar dicho mensaje.
* generateRandomId: función que retorna un número de ID aleatorio.
* isPersisted: esta función retorna **true** o **false** dependiendo de si el agente se encuentra persistido en la base de datos o no.
* clone: dicha función retorna un nuevo agente, clon de él mismo, pero con transactionNumber en 0, un nuevo identificador único para este nuevo agente y sin chainId.
* canSupplyRequest:
* proccessCallForProposal:
* processAgree:
* processCancel:
* processPropose:
* processAcceptProposal:
* processRejectProposal:

### chain.js



En este archivo se encuentra el desarrollo de la clase **Chain**, la encagada del manejo de las cadenas de suministros. Las mismas pueden ser de tipo **template** o no, lo que indica que esa cadena sólo será usada como plantilla para otras cadenas de suministros. Que una cadena sea de tipo **template**, implica que a los agentes que la conforman no será posible enviarles mensajes.

#### Atributos de la clase **Chain**

La clase **Chain** presenta las siguientes variables de clase:

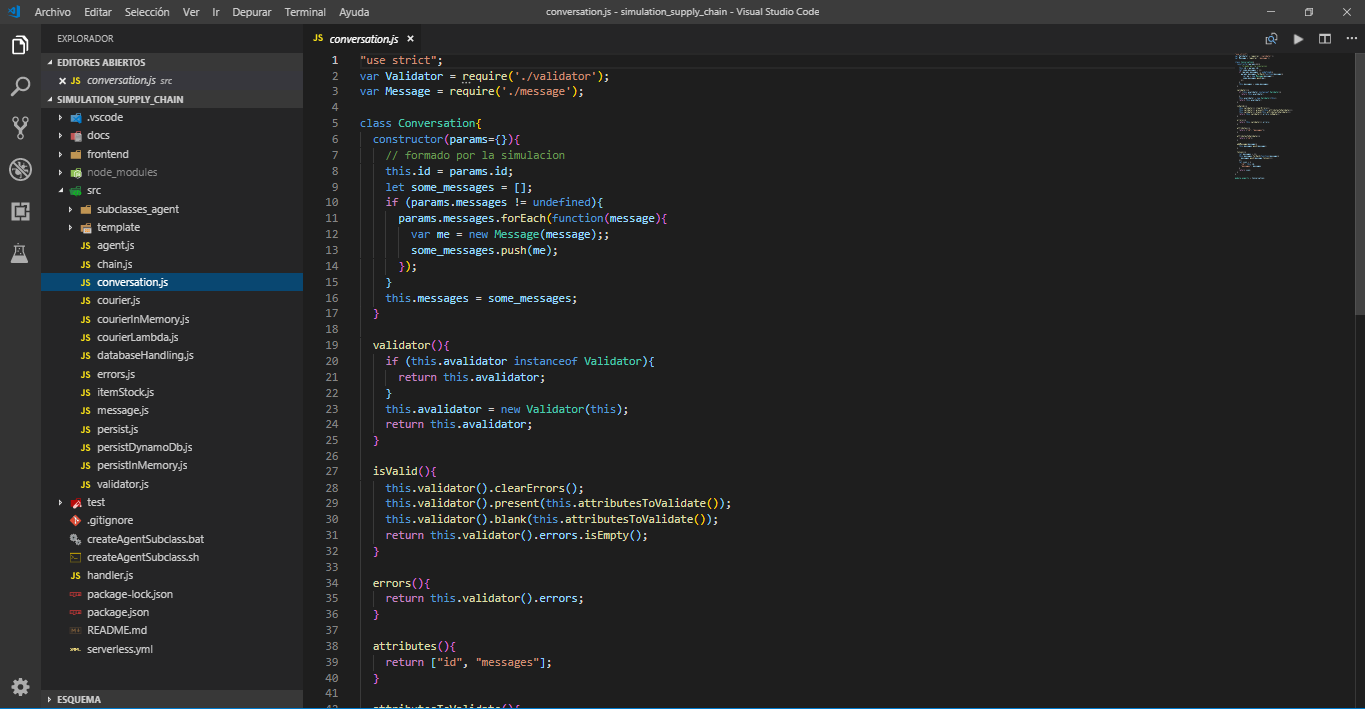
* id: (int) representa el identificador único de la clase.
* chainName: (string) representa el nombre de la cadena de suministros.
* description: (string) representa una breve descripción de la cadena de suministros.
* template: (boolean) indica si la cadena es un **template** o no.
* transactionNumber: (int) similar al mismo atributo presente en la clase **Agent**.
* agents: (Array<**Agent**>) colección de agentes que componen la cadena de suministros.

#### Métodos de la clase **Chain**

Dentro de la clase **Chain** se encuentran los siguientes métodos:

* constructor: recibe un hash con los atributos básicos de una cadena y los instancia.
* validator: utiliza el patrón **Singleton** para instancia un objeto del tipo **Validator.**
* isValid: utilizando el objeto validador instanciado por la función anterior, indica si la cadena es válida o no.
* errors: utilizando el objeto validador antes descripto, retorna los errores encontrados por el mismo.
* attributes: función que retorna un arreglo con las variables de clase de la clase **Chain**.
* attributesToValidate: a diferencia de la función anterior, esta devuelve un arreglo con las variables que se validarán en la clase **Chain**.
* toJson: función que retorna un hash con las variables de clase de la cadena que serán enviadas en una petición HTTP o hacia la base de datos.
* generateRandomId: función que retorna un número de ID aleatorio.
* updateAttributes: dicho método recibe un hash con variables y sus valores y actualiza las variables de clase de la cadena.
* isPersisted: esta función retorna **true** o **false** dependiendo de si la cadena se encuentra persistida en la base de datos o no.
* clone: dicha función retorna una nueva cadena, clon de ella misma, pero con el mismo chainName concatenado al sufijo “ copy”, transactionNumber en 0, un nuevo identificador único para esta nueva cadena, no es template y sus agentes son clones de los agentes de la cadena original.
* clearConversations: este método elimina las conversaciones de todos los agentes que conforman la cadena de suministros.

### conversation.js



Un conjunto de mensajes conforma una conversación y la misma es representada por la clase **Conversation**.

#### Atributos de la clase **Conversation**.

Esta clase posee las siguientes variables de clase:

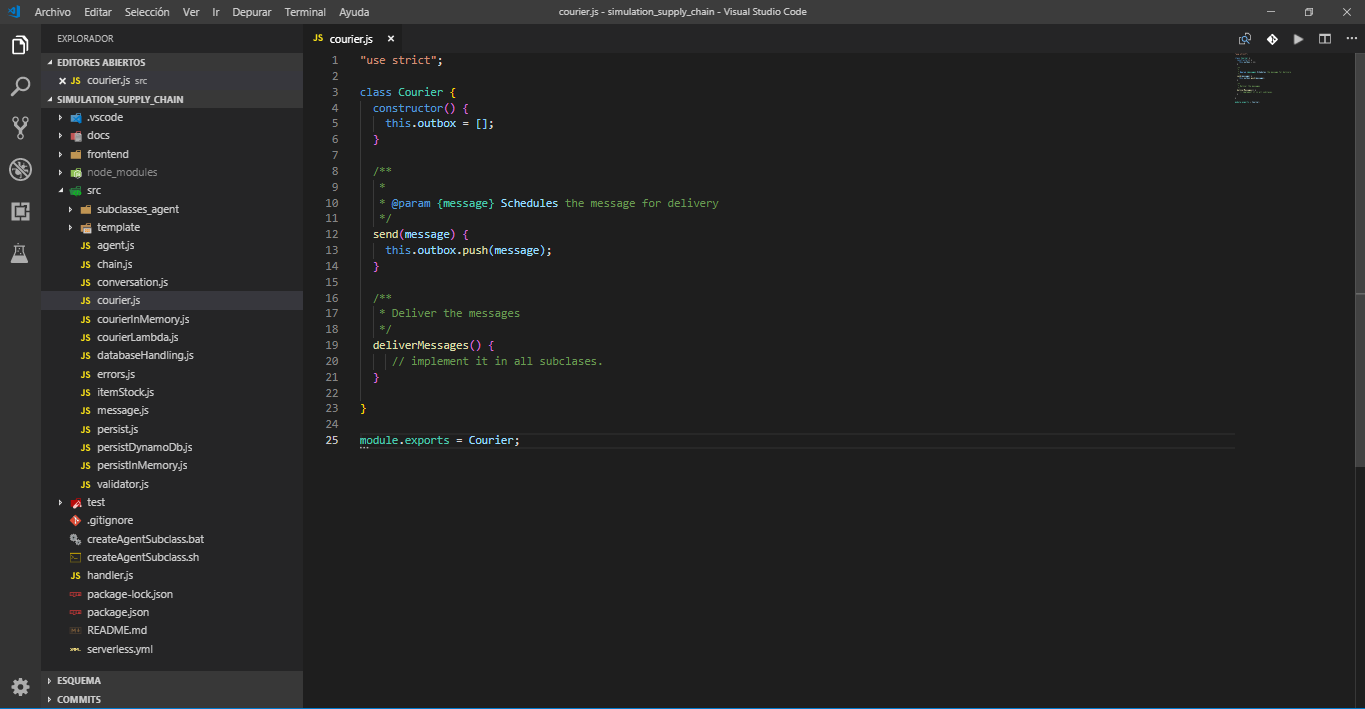
* id: (int) representa el identificador único de la clase.
* messages: (Array<**Message**>) arreglo de mensajes que conforman la conversación.

#### Métodos de la clase **Conversation**.

**Conversation** utiliza los siguientes métodos:

* constructor: recibe un hash con los atributos básicos de una conversación y los instancia.
* validator: utiliza el patrón **Singleton** para instancia un objeto del tipo **Validator.**
* isValid: utilizando el objeto validador instanciado por la función anterior, indica si la conversación es válida o no.
* errors: utilizando el objeto validador antes descripto, retorna los errores encontrados por el mismo.
* attributes: función que retorna un arreglo con las variables de clase de la clase **Conversation**.
* attributesToValidate: a diferencia de la función anterior, esta devuelve un arreglo con las variables que se validarán en la clase **Conversation**.
* addMessage: este método recibe un mensaje por parámetro y lo agrega a su colección de mensajes.
* toJson: función que retorna un hash con las variables de clase de la conversación que serán enviadas en una petición HTTP o hacia la base de datos.

### courier.js



La clase **Courier** es una clase abstracta. Su finalidad es la de enviar los mensajes entre los diferentes agentes que conforman la cadena de suministros.

#### Atributos de la clase **Courier**

Esta clase sólo cuenta con un atributo de clase:

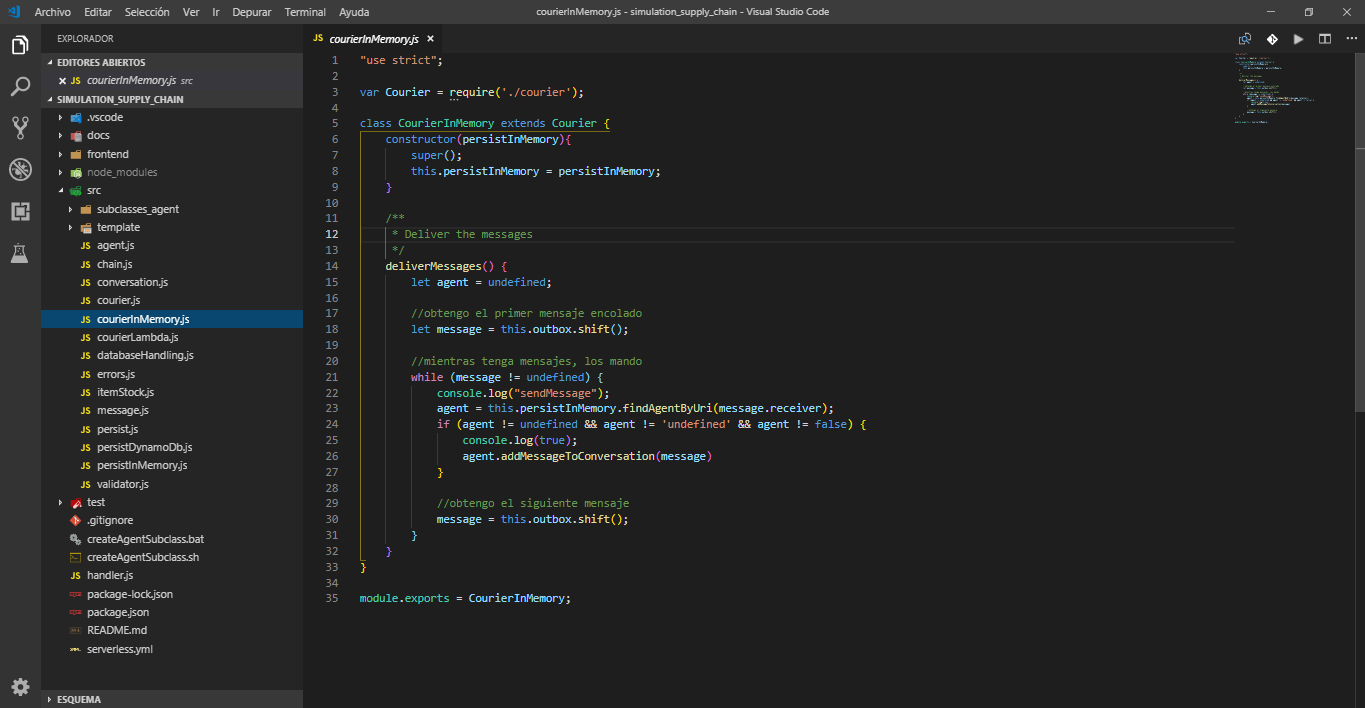
* outbox: (Array<**Message**>) colección de mensajes a enviar.

#### Métodos de la clase **Courier**

La clase **Courier** cuenta con los siguientes métodos:

* constructor: instancia el atributo outbox vacío.
* send: este método recibe un mensaje por parámetro y lo guarda en su colección de mensajes.
* deliverMessage: método abstracto que debe ser implementado en cada subclase de **Courier**. El mismo será el encargado de enviar los mensajes guardados en outbox.

### courierInMemory.js



Esta clase es subclase de **Courier**. La particularidad que tiene esta subclase es que está definida para enviar mensajes a agentes “en memoria”, es decir objetos JavaScript instanciados en su misma ejecución. Esta clase es útil para realizar simulaciones en forma local, sin salir a un entorno externo como puede ser internet.

#### Atributos de la clase **CourierInMemory**

Esta clase, además del atributo heredado de su superclase, cuenta con el siguiente atributo de clase:

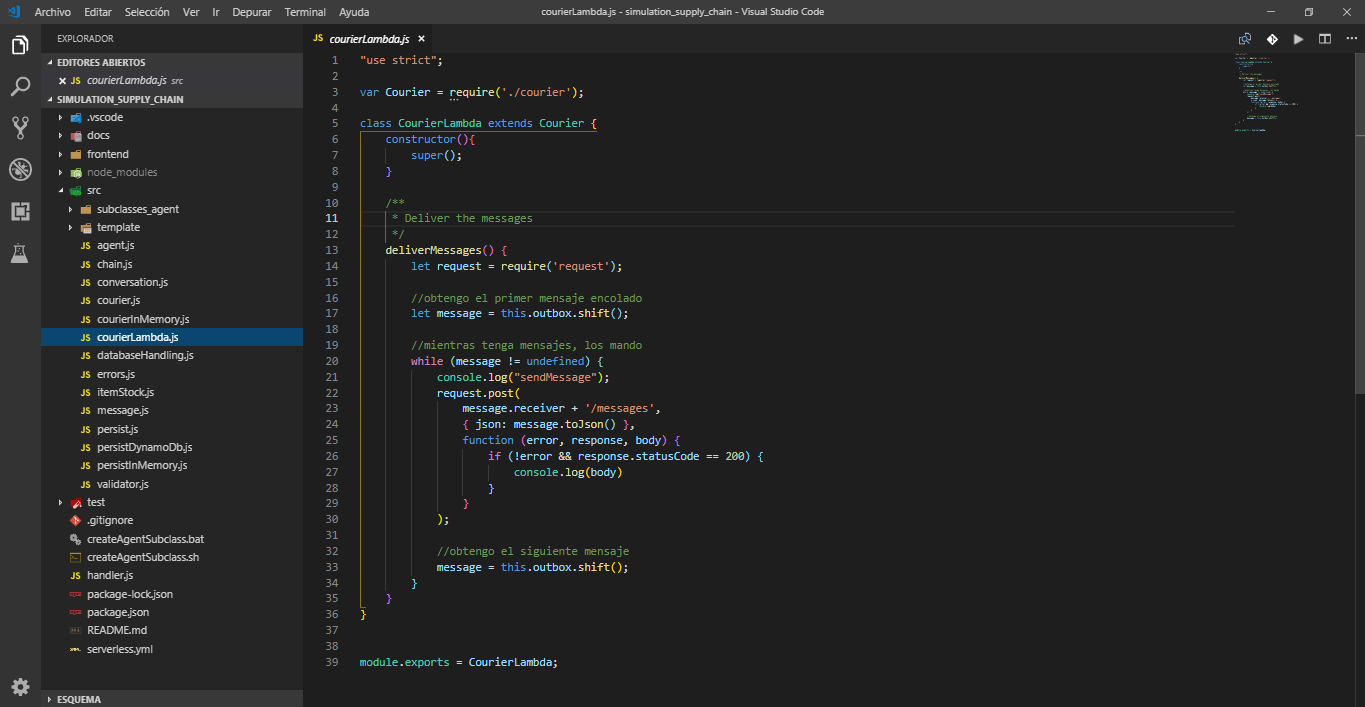
* persistInMemory: (**PersistInMemory**) objeto de la clase **PersistInMemory** utilizado para manejar los agentes en memoria.

#### Métodos de la clase **CourierInMemory**

La clase **CourierInMemory** implementa el método abstracto de su superclase:

* constructor: recibe por parámetro un objeto **PersistInMemory** e instancia con él su variable de clase.
* deliverMessages: mientras tenga mensajes para enviar, este método los obtiene uno por uno, le pide al objeto **PersistInMemory** el agente destinatario del mensaje y le dice a dicho agente que guarde, entre sus conversaciones, este mensaje.

### courierLambda.js



Esta clase es subclase de **Courier**. La particularidad que tiene esta subclase es que está definida para enviar mensajes a agentes que no se encuentran “en memoria”. Esta clase es útil para realizar simulaciones en forma remota, saliendo a un entorno externo, internet.

#### Atributos de la clase **CourierLambda**

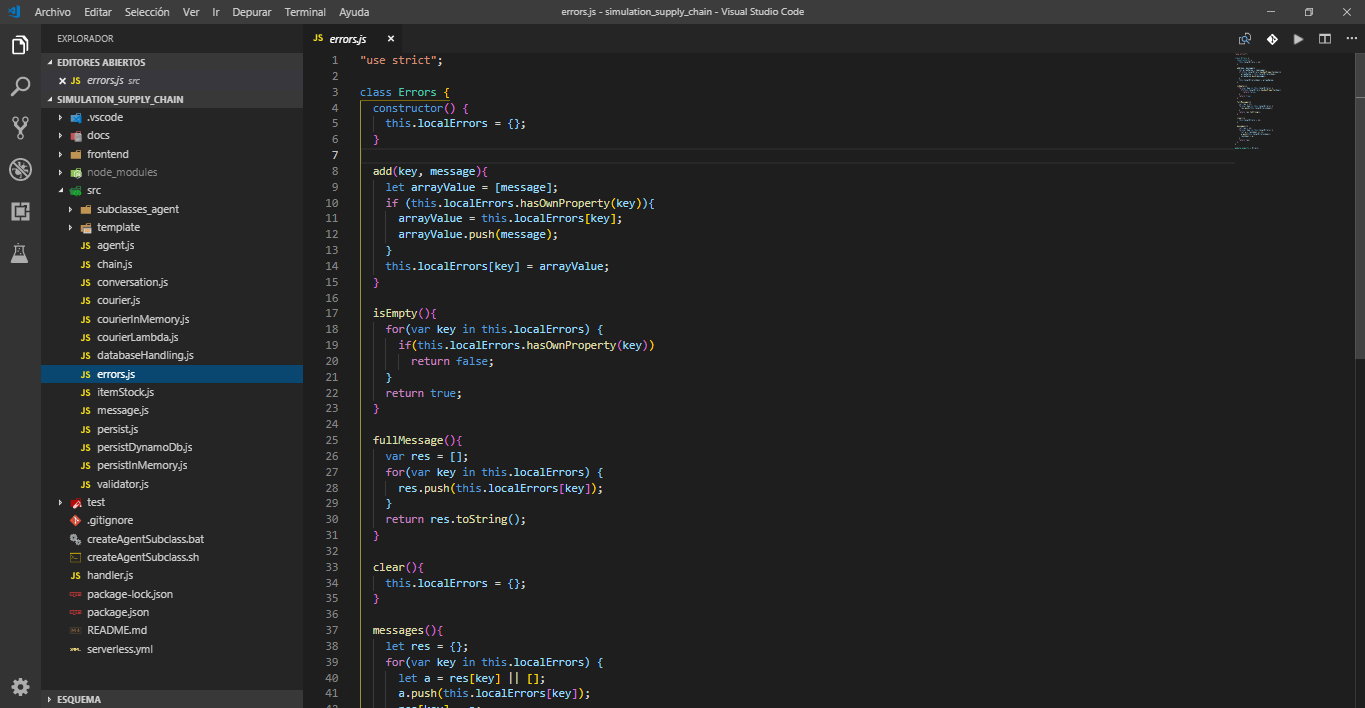
Esta subclase de **Courier**, a diferencia de **CourierInMemory**, solo cuenta con el atributo heredado de su superclase.

#### Métodos de la clase **CourierLambda**

Al igual que **CourierInMemory**, **CourierLambda** también implementa el método abstracto de su superclase:

* deliverMessages: mientras tenga mensajes para enviar, este método los obtiene uno por uno, genera una petición POST con el mensaje y lo envía a su destino.

### errors.js



La clase **Errors** es utilizada por la clase **Validator** para almacenar y manejar los errores encontrados durante las validaciones de los objetos.

#### Atributos de la clase **Errors**.

Esta clase cuenta con el siguiente atributo:

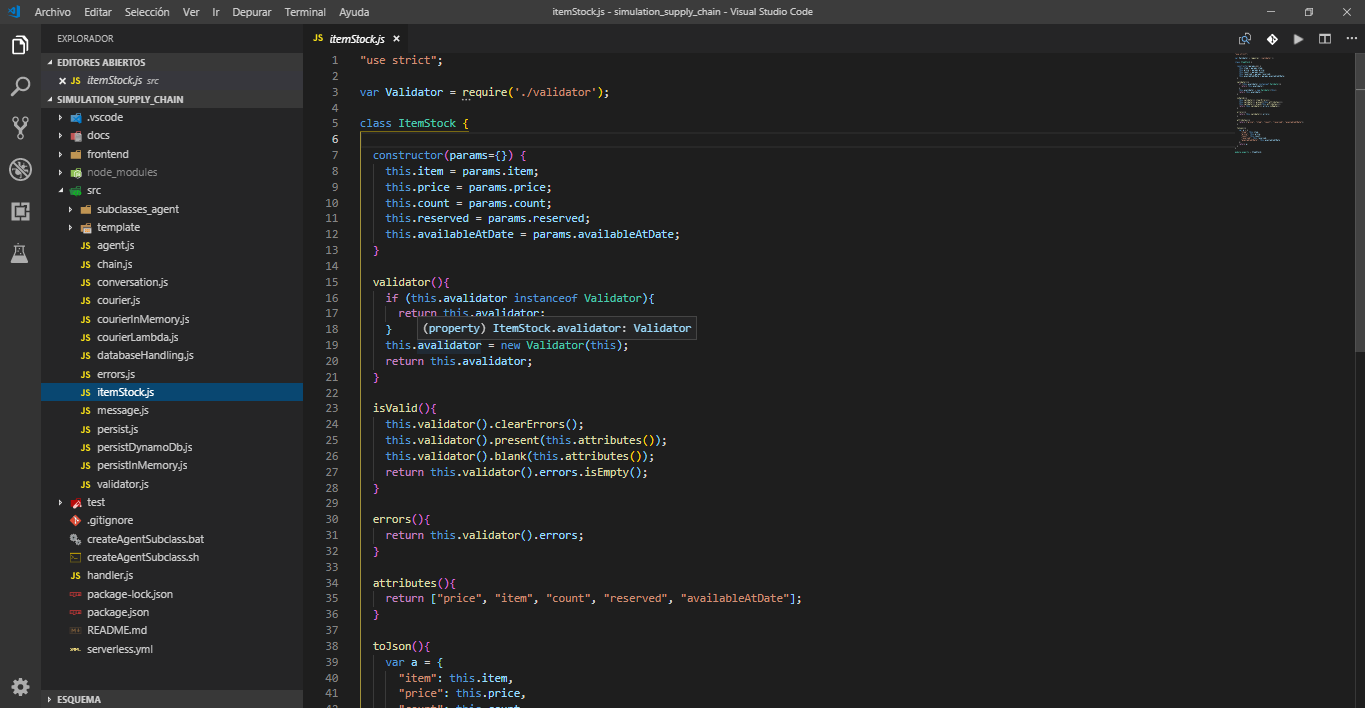
* localErrors: (Hash<string,string>) este atributo es un hash que por clave lleva el atributo que causó el error y como valor lleva el mensaje de error explicativo.

#### Métodos de la clase **Errors**

Los métodos de la clase **Errors** son los siguientes:

* constructor: instancia su variable de clase vacía.
* add: este método recibe por parámetro una clave y un valor para su hash de errores. Transforma el mensaje en un arreglo, evalúa si ya poseía la clave enviada dentro del hash, en caso positivo, encola el nuevo mensaje al arreglo de mensajes de la clave enviada por parámetro, en caso negativo, lo agrega el mensaje, transformado en arreglo, bajo la clave recibida.
* isEmpty: esta función indica si el atributo localErrors almacena algún error o no.
* fullMessage: la función mencionada retorna todos los mensajes de error almacenados en la variable de clase localErrors concatenados en un string.
* clear: vacía el hash de errores de la clase.
* messages: esta función retorna todos los mensajes de error almacenados en la variable de clase localErrors como un hash de atributo/mensajes.

### itemStock.js



La clase **ItemStock** es la encargada de representar el stock que tiene un agente de cierto ítem que negocia con otros agentes.

#### Atributos de la clase **ItemStock**

Esta clase presenta los siguientes atributos:

* ítem: representa al ítem que negocia con otros agentes.
* price: esta variable consiste en el precio por unidad del ítem del agente.
* count: dicho atributo representa la cantidad total del ítem del agente.
* reserved: esto representa la cantidad de ítem que tiene reservado para otro/s agente/s.
* availableAtDate:

#### Métodos de la clase **ItemStock**

Los métodos que utiliza esta clase son los siguientes:

* constructor: recibe un hash con los atributos básicos de un **ItemStock** y los instancia.
* validator: utiliza el patrón **Singleton** para instancia un objeto del tipo **Validator.**
* isValid: utilizando el objeto validador instanciado por la función anterior, indica si el **ItemStock** es válido o no.
* errors: utilizando el objeto validador antes descripto, retorna los errores encontrados por el mismo.
* attributes: función que retorna un arreglo con las variables de clase de la clase **ItemStock**.
* toJson: función que retorna un hash con las variables de clase del **ItemStock** que serán enviadas en una petición HTTP o hacia la base de datos.

### message.js