Asking questions away with Inquirer!

Edit New Page Jump to bottom

Paul Fitzgerald edited this page on 12 Jul 2018 · 3 revisions

> We get wise by asking questions, and even if these are not answered, we get wise, for a well-packed question carries its answer on its back as a snail carries its shell. - James Stevens

Have you ever used the `npm init` command? If not, you can read about it here. It's an easy command. It asks you some questions and it will create a package.json file for you to get you started out on your node modules.

Also, if you have worked with the `Yeoman` system, and you run `yo` to scaffold a project out, then you know that yo lists out generators available and lets you choose a generator.

Maybe, you want to ask questions like this command, in your command line tool.

Well, then, you can use Inquirer.js. The `yo` cli uses `inquirer`.

Inquirer.js is a framework and a way we can ask our users, questions, at the command line in our node projects. You create your questions, and you use inquirer's api to ask these questions and the answers are made available to you. And you can do something with them. You can parse/validate the answers, go do something based on the responses, for instance.

So the Inquirer system is built on the model of `questions` and `answers`. Two simple object models.

Getting started with Inquirer.js

Let's see how to use this question/answer framework.

```
var inquirer = require('inquirer');
console.log(inquirer);
```

```
{ prompts: {},
  Separator: { [Function: Separator] exclude: [Function] },
  ui:
   { BottomBar: { [Function: Prompt] super_: [Function: UI] },
     Prompt: { [Function: PromptUI] super_: [Function: UI] } },
  createPromptModule: [Function],
  prompt:
   { [Function]
     prompts:
      { list: [Object],
        input: [Object],
        confirm: [Object],
        rawlist: [Object],
        expand: [Object],
        checkbox: [Object],
        password: [Object] },
     registerPrompt: [Function],
     restoreDefaultPrompts: [Function] },
  registerPrompt: [Function],
  restoreDefaultPrompts: [Function] }
```

Inquirer has a pretty simple object exported. `Seperator` , `ui` , `prompt` . `prompt` is a method and that's what we are going to use first.

To ask questions, we create question objects and put them in an array, and pass it to the prompt function. We can pass a callback function to prompt.

After all the questions are asked, the callback function is called by Inquirer.js

Inquirer.js records the answers in a neat object structure for us to look at later and passes it as a parameter to the callback.

```
inquirer.prompt([], function(answers){
});
```

or better ...

```
function doSomething(answers){
  // Do whateva you want!
}
var questions = [];
inquirer.prompt(questions, doSomething);
```

That's so easy! Now all we need to do is create some questions.

The question object model

To do that, we need to understand the `question` object structure. It's an easy object model. Docs can be found at https://github.com/SBoudrias/Inquirer.js#question

property	purpose
message	The question that will be displayed
type	What kind of a question is this?
name	When inquirer stores our answers, how do we know that the answer is for this question

property	purpose
default	Sometimes, you want to provide default answers.
choices	An array of choices for the user to choose from. It can be a function that returns this array as well.
validate	User inputs sometime need to be validated. You might expect a number for an age, but the user might have entered a string.
filter	Given the user selection, we can do something with it by adding a function.

Question types

With Inquirer, we can ask a variety of questions.

type	how it works
input	user can use the keyboard and enter text
list	user can use `up` and `down` arrow keys and select an option from a list of options
rawlist	user can enter an option number or key instead of navigating the list
checkbox	user can select multiple options from a list of options
confirm	user can specify yes or no
password	User inputs text, but all text is displayed as *s on the screen

Creating some questions

Let us actually create some questions.

We can find a lot of examples in the Inquirer's examples folder.

Here, however, we would like to model a user subscription list based questions here.

We would like to know a user's first and last name, email id, age, subscription interests(programming languages in our case), best role description, a feedback rating(1-5) and we finally want to confirm if we submit these details.

Let's start with one question first. And we shall keep adding to the list of questions.

```
var inquirer = require('inquirer');

function processAnswers(answers){
  console.log("And your answers are:", answers);
}
var questions = [
{
    message: "What's your first name?",
    type: "input",
    name: "firstName"
}];
inquirer.prompt(questions, processAnswers);
```

When we run the script using `node index.js` command, we are asked the first name.

```
$ node index.js
? What's your first name? Sameeri
And your answers are: { firstName: 'Sameeri' }
```

Just to test it out, i ran the command again, and this time did not provide an answer.

```
$ node index.js
? What's your first name?
And your answers are: { firstName: '' }
```

An empty response is still valid. We should maybe check for an empty string and make this a `required` .

To do this is simple, adding a `validate` property and attaching a validation method will do the trick.

```
var inquirer = require('inquirer');

function processAnswers(answers){
  console.log("And your answers are:", answers);
}
var questions = [
{
    message: "What's your first name?",
    type: "input",
    name: "firstName",
    validate: function validateFirstName(name){
        return name !== '';
    }
}];
inquirer.prompt(questions, processAnswers);
```

Now, when we run the command, we are presented with the same question. However, not answering it is a no-no. How sweet!

```
$ node index.js
? What's your first name?
>> Please enter a valid value
```

From now on, we will just look at the question structures. Since, only the question structures are changing.

Let's ask the last name now. It's similar.

```
{
    message: "What's your last name?",
    type: "input",
    validate: function validateLastName(name){
        return name !== '';
    }
}
```

When i run the script again, i am thrown an ugly error.

```
Error: You must provide a `name` parameter
```

Apparently i forgot the name property on my latest question. We need a `name` property. Let's fix it.

The name property is a must since it is used in the storing of `answers` . This can be observed in our `answers` object.

```
$ node index.js
? What's your first name? Sameeri
? What's your last name? Marryboyina
And your answers are: { firstName: 'Sameeri', lastName: 'Marryboyina' }
```

Answers object model

The answers provided are stored in a plain object format(name-value) pair. The name is the `name` provided in the question.

```
{
"firstName" : "Sameeri",
"lastName" : "Marryboyina",
}
```

values are different, and are based on the question type.

question type	data type	value
input	String	user input
confirm	Boolean	true or false
list	String	user selection

question type	data type	value
rawlist	String	user selection
checkbox	Array	user selections

```
var inquirer = require('inquirer');

function processAnswers(answers){
  console.log("And your answers are:", answers);
}

function validateName(name){
        return name !== '';
    }

var questions = [
{
    message: "What's your first name?",
    type: "input",
    name: "firstName",
    validate: validateName
},{
    message: "What's your last name?",
    type: "input",
    name: "lastName",
    validate: validateName
},

];
inquirer.prompt(questions, processAnswers);
```

Adding number based questions

Our next question in the list is finding the user's age.

```
function validateAge(age)
{
   var isValid = !_.isNaN(parseFloat(age));
   return isValid || "Age should be a number!";
}

var ageQ = {
    message: "What's your age?",
    type: "input",
    name: "age",
    validate: validateAge
};
```

When we interact with this question at the command line

```
$ node index.js
? What's your first name? Sameeri
? What's your last name? Marryboyina
? What's your age? 219928kjsakj
And your answers are: { firstName: 'Sameeri',
  lastName: 'Marryboyina',
  age: '219928kjsakj' }
```

Wierd! Well, it's not, actually. Small mechanics here.

Our validation code does this piece of test : `parseFloat("219928kjsakj")` .

This would return the number 219928.

This is not a `NaN` . So it's fine, the function returns `true` .

A signal is sent to inquirer, that this answer is valid, please store the response.

Well, inquirer stores the input as is.

Since we only want integers, we can just use a simple regular expression `/^\d+$/` .

Now our validateAge method looks very simple and it only allows integers.

```
function validateAge(age)
{
   var reg = /^\d+$/;
   return reg.test(age) || "Age should be a number!";
}
```

Awesomeness. But the number entered can be 21298, which might not be a valid age.

Validating with Joi

We can see that our validation logic can quickly get out of control. Also, we would be creating validation code that will be repeated over many projects. This is totally unnecessary. A better solution is to use an existing validation framework.

Joi is an amazing validation framework.

As soon as you look at the API, you can see the power. We would like to incorporate such power into our questions.

```
var joi = require('joi');
joi.validate(objectToValidate, schemaDefinition, function(err, val){
});
```

joi supports complex object validation. You specify the schema and pass in the object to validate, the schema against which the validation needs to be made, and a callback.

With inquirer, we are dealing with simple strings at a time, so we can make validations against simple types.

Also it's important to realize that inquirer expects a `true` if the answer is valid. If it gets a false, it will continue asking the same question. If it gets a string, it thinks that this is an error message. So it displays that message.

Armed with this knowledge, we can redo our validation of names.

```
function validateName(name) {
        var valid;
        Joi.validate(name, Joi.string().required(), function(err,val){
            if (err){
                console.log(err.message);
                valid = err.message;
            }
            else{
                valid = true;

            }

        });
        return valid;
}
```

Simple. we want to pass in the name and validate against a `Joi.string().required()` schema.

If there is an error, we want to return this.

Let's try and work with our validateAge function now. It would be similar code.

```
function validateAge(age) {
        var valid;
        Joi.validate(age, Joi.number().required().min(10).max(99), function(err,val
            if (err){
                console.log(err.message);
                valid = err.message;
            }
            else{
                valid = true;

            }

        });
        return valid;
}
```

The only thing that changed here is our schema definition, and yeah the variable name called age.

Let's try to refactor this code duplication.

We are using an unnecessary `valid` variable. We are using it to store either true or the error message. Instead we can return true or error message and return it from the function the same value.

```
function validateName(name) {
       return Joi.validate(name, Joi.string().required(), function(err,val){
           if (err){
               console.log(err.message);
               return err.message;
           }
           else{
              return true;

           }

       });
}
```

We could also move the `anonymous function` out and name it. Since this is the repeated code.

```
function onValidation(err,val){
    if(err){
        console.log(err.message);
        return err.message;
    }
    else{
        return true;
    }

}

function validateName(name) {
       var schema = Joi.string().required();
       return Joi.validate(name, schema, onValidation);
}

function validateAge(age) {
       var schema = Joi.number().required().min(10).max(99);
       return Joi.validate(age, schema , onValidation);
}
```

That's so better. Our functions are much smaller and it's easy to ready the schema.

Also, the `callback` function is `optional`.

We could have as well written the code without a callback as such.

```
function validateAge(age) {
       var schema = Joi.number().required().min(10).max(99);
       var result = Joi.validate(age, schema);
       return result.err ? result.err.message : true;
}
```

If we had the context of the question object inside the validate methods, we can write a generic function and store the schemas as an object. That would be more awesome. But sadly, i could not figure out how to.

Let's continue adding our questions. :)

Next, we want an email. This is going to be super simple using joi.

```
function validateEmail(email) {
       return Joi.validate(email, Joi.string().email(), onValidation);
}

var emailQ = {
    message: "What's your email?",
    type: "input",
    name: "email",
    validate: validateEmail
};
```

```
$ node index.js
? What's your first name? Sameeri
? What's your last name? Marryboyina
? What's your age? 31
? What's your email? sam
"value" must be a valid email
? What's your email? sam@
? What's your email? sameeri@smarryboyina.com
And your answers are: { firstName: 'Sameeri',
  lastName: 'Marryboyina',
  age: '31',
  email: 'sameeri@smarryboyina.com' }
```

Multiselect questions

We need to be able to ask which topics the user is interested in.

```
var topicsQ = {
        message: "What topics would you like to subscribe to?",
        type: "checkbox",
        name: "topics",
        choices: ["JavaScript", "C#", "Ruby", "Java", "Python"]
    }
```

That would bring up something like this. User can select multiple items off this list.

```
$ node index.js
? What's your first name? Sameeri
? What's your last name? Marryboyina
? What's your age? 32
? What's your email? sm@marryboyina.com
? What topics would you like to subscribe to? (Press <space> to select)
>(x) JavaScript
 (x) C#
 ( ) Ruby
 ( ) Java
 ( ) Python

And your answers are: { firstName: 'Sameeri',
  lastName: 'Marryboyina',
  age: '32',
  email: 'sm@marryboyina.com',
  topics: [ 'JavaScript', 'C#' ] }
```

This is neat. What if we wanted to have JavaScript selected by default. One or more things are initially selected.

This can be done by passing an array of objects to the `choices` property instead of a simple string array.

```
var topicsQ = {
        message: "What topics would you like to subscribe to?",
        type: "checkbox",
        name: "topics",
        choices: [{name: "JavaScript", checked: true},
                  {name: "C#", checked: false},
                  {name: "Java"},
                  {name: "Ruby"},
                  {name: "Python", checked:false} ]
    };
```

We can make sure that atleast one item is selected by adding a validation method.

```
function validateTopics(topics){
     var schema = Joi.array().min(1);
     return Joi.validate(topics, schema, onValidation);
}
```

Single item selection lists

Now we can go ahead and look at two types of lists - `list` and `rawlist` . Both allow the user only to select one item. The way user interacts with it is different.

```
var roleDescriptionQ = {
        message: "How would you best describe yourself?",
        type: "list",
        name: "role",
        choices: ["Beginner", "Intermediate", "Coding Ninja" ]
    };
```

When we are presented the question, we have an interface like this. User can use the `up` and `down` arrow keys and select an option.

```
? How would you best describe yourself?
> Beginner
  Intermediate
  Coding Ninja
```

```
var feedbackQ = {
        message: "How would you rate us?",
        type: "rawlist",
        name: "feedback",
        choices:["Awesome", "Good", "Okay", "You suck" ]
    };
```

The rawlist question interface is a little different.

```
? How would you rate us?
  1) Awesome
  2) Good
  3) Okay
  4) You suck
  Answer:
>> Please enter a valid index
```

You get to provide an answer here. If it's anything else than the numbers displayed, you get an error.

The answers object is constructed as such:

```
And your answers are: { firstName: 'Sam',
  lastName: 'ajas',
  age: '21',
  email: 'sa@sak',
  topics: [ 'JavaScript' ],
  role: 'Beginner',
  feedback: 'Good' }
```

Conditional questions

Let's say a user gives us an "Awesome" rating. We want to give them a treat. We will give them either our magnet and sticker, t-shirt or mug. By default, we will give a t-shirt.

We can phrase such questions very easily by including a `when` property on the question object.

```
var treatQ = {
        message: "Thank you soo much! You are Awesome too...",
        type: "list",
        name: "treat",
        choices: ["Magnet&Stickers","T-Shirt", "Mug"],
        when : function( answers ) {
                return answers.feedback === "Awesome";
              },
        default: "T-Shirt"
    };
```

We can specify a function for the the `when` property that can get the `answers` object from inquirer. Also, the default item in the list is the first item in the `choices` array. We can override that by specifying a `default` property.

Confirmation questions

Our final question will be a submit question. Confirm returns a true or false. And we would like our default answer to be true in this case.

```
var subscribeQ = {
        message: "Would you like to go ahead and subscribe to our newsletter?",
        type: "confirm",
        name: "subscribe"
        default: true
    };
```

```
$ node index.js
? What's your first name? Sameeri
? What's your last name? Marryboyina
? What's your age? 21
? What's your email? sam@sam.com
? What topics would you like to subscribe to? JavaScript, Ruby
? How would you best describe yourself? Intermediate
? How would you rate us? Awesome
? Thank you soo much! You are Awesome too... Magnet&Stickers
? Would you like to go ahead and subscribe to our newsletter? Yes
And your answers are: { firstName: 'Sameeri',
  lastName: 'Marryboyina',
  age: '21',
  email: 'sam@sam.com',
  topics: [ 'JavaScript', 'Ruby' ],
  role: 'Intermediate',
  feedback: 'Awesome',
  treat: 'Magnet&Stickers',
  subscribe: true }
```

There you go! We have created a bunch of questions, validated some user inputs using joi, provided defaults, provided conditional based questions, created a multiselect based question and built a complete subscription form.

What we actually want to do with the answers depends on the context we are going to use inquirer in.

+ Add a custom footer

Pages 2

Find a Page...

Home

Asking questions away with Inquirer!

+ Add a custom sidebar

Clone this wiki locally

```
https://github.com/sameeri/Code-Inquirer.wiki.git
```