Organic Development

Krasimir Tsonev on Jul 3rd 2013 with 20 Comments

Tutorial Details

-
- **Difficulty**: Intermediate
- **Estimated Completion Time**: 30 minutes

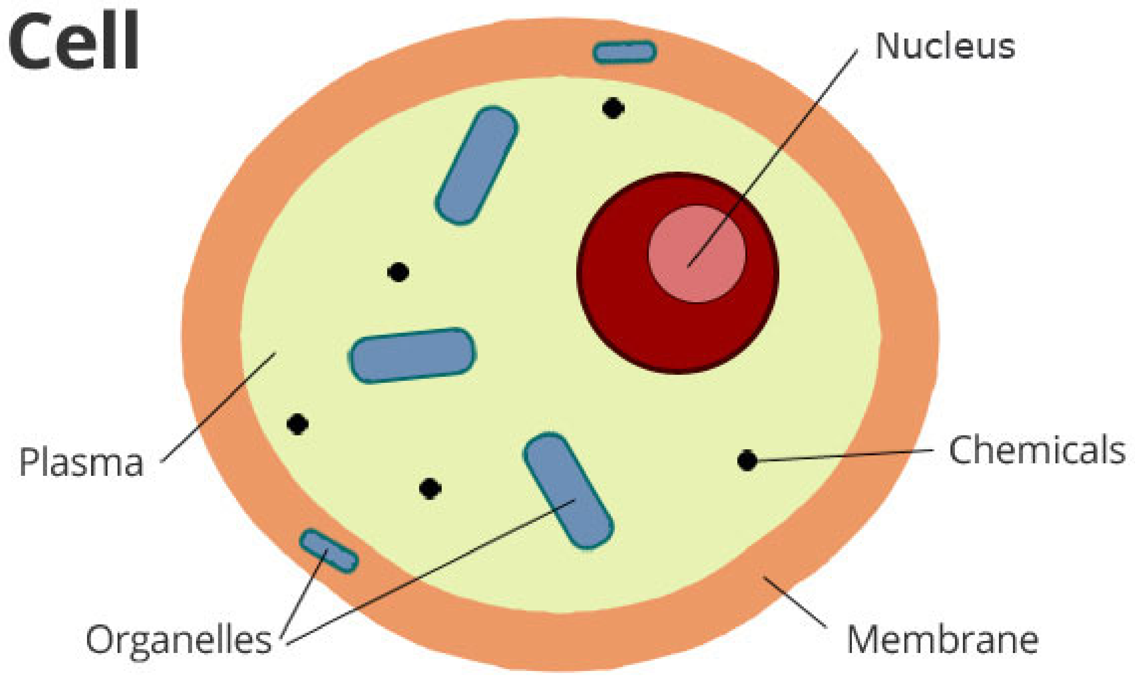
View post on Tuts+ Beta**Tuts+ Beta** is an optimized, mobile-friendly and easy-to-read version of the Tuts+ network.

Introduction

I was working as a graphic designer a few years ago and a common problem that I would run into was picking color schemes for new projects. One of my colleagues said, *"Just pick a nice photo and grab colors from there"*. This technique works well because photos offer you a natural combination of colors. So I was thinking, "Why not transfer this same concept to my work as a coder?". And this is where Organic comes in to play. When I was first introduced to Organic I was amazed how simple it was and at the same time, how flexible its approach is. Finally, I had something which encourages modular programming, it's just as useful as the MVC pattern, and it's a great tool for architecting.

The Concept

As you may have guessed, the Organic concept is biology based. Your main application acts as a *Cell*, which has a *Membrane* and a *Nucleus*. But the real job of a Cell is done by the *Organelles*, which communicate between each other with *Chemicals*. Of course, the elements and the processes in Organic are not 100% identical to real life Cells, but they are pretty close. Now, I know it sounds crazy, but once you start working with it you'll see how simple and natural this approach can be when applying it to your apps.



Download Organic

Organic is distributed as a Node module. So you should have NodeJS already installed. If you don't, please go to nodejs.org and grab the latest version for your

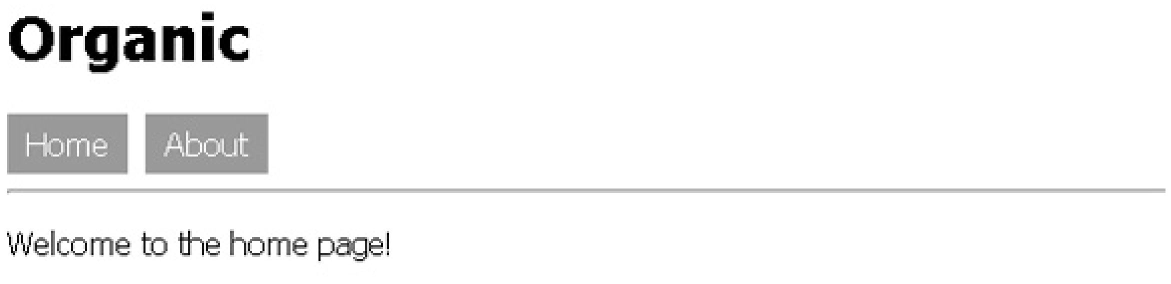
OS. Your `package.json` file should look like this:

```
{
    "name": "OrganicDevelopment",
    "version": "0.0.0",
    "description": "Organic development",
    "dependencies": {
        "organic": "0.0.11"
    },
    "author": "Your Name Here"
}
```

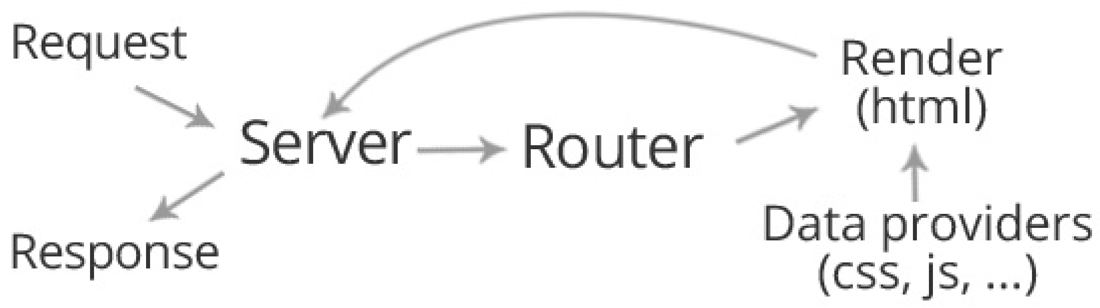
Run `npm install` in the same directory and the manager will download the necessary files. The core of Organic is actually pretty small. It contains only the definition of the main elements – Cell, Nucleus, Membrane, Plasma, Organelle, Chemical, and DNA. Of course it comes with a few tests, but it's a small package overall. This helps in making it easy to learn and start developing with almost immediately.

The Example

For this article I decided to create a simple web site using only the core of Organic. The source code can be downloaded at the top of this article, if you'd like to follow along. I think that this sample application is the best way to present this new pattern. The site contains two pages – `Home` and `About`. Here's a screenshot of the site:



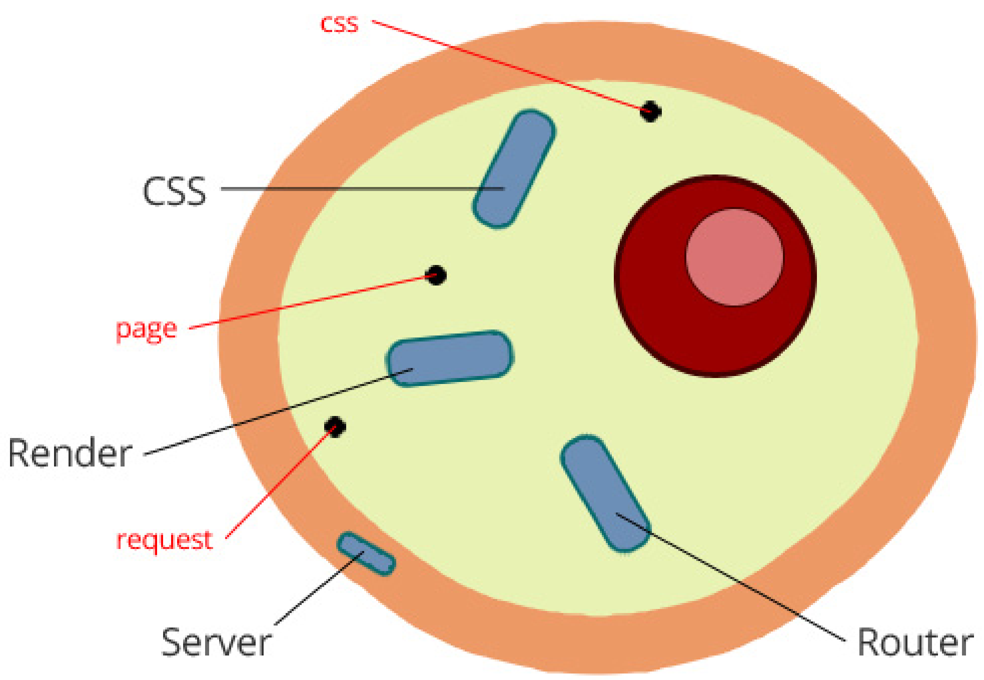
The app contains two buttons linking to the two different pages. The `About` page has just a little bit more text than the `Home` page does. Simple enough, but let's see what's behind the curtains. Here's a diagram displaying the basic request flow of our application:



The user sends a request to our NodeJs application. The Server accepts the request and sends it to the Router. After that, the Render knows which page should be used and returns an answer to the Server. At the end, the response is then sent to the user.

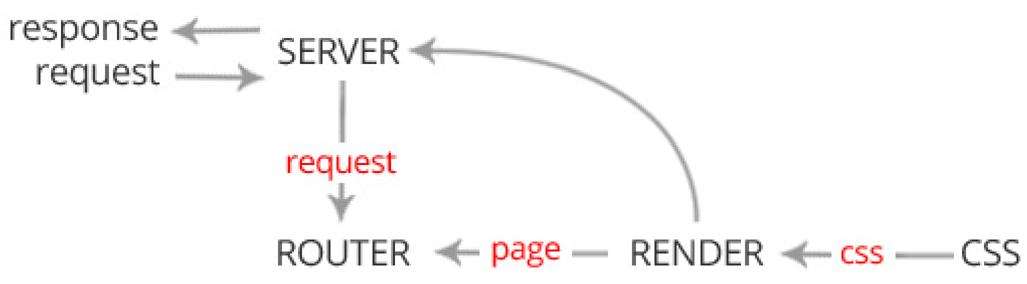
There is one additional element, Data Providers, which prepares the needed CSS or JavaScript for the Render (keep in mind that in our example app I didn't use JavaScript, there is only a CSS module).

Here's what our app would look like as a Cell, in Organic:



In the Cell, we have a membrane which keeps the internal elements away from the outside world. Inside of this membrane is where we'll put our first organel, our Server, because this is where data can either enter or leave our application. The other organelles (Router, Render, and CSS) are placed in the plasma. All of these modules are communicating with each other via chemicals (*request*, *page* and *css*, marked in red). The Server emits a *request* chemical. The Router emits a *page* and the CSS organel sends the *css*. I should also mention that the plasma acts as an event bus for the chemicals. Organelles listen for a particular chemical and if found, they react on it.

Here's another request flow diagram, but this time with the chemicals that are emitted (marked in red):



Now if this concept is still unclear to you, don't worry, as we proceed through the next few sections and get into the actual code, it should begin to make more sense!

DNA

Everything starts with the DNA (Deoxyribonucleic acid), which you can think of as a Cells configuration. This DNA is where you will define your organelles and their settings.

Let's create a new `index.js` file and put in the following code:

```
var DNA = require("organic").DNA;
var Cell = require("organic").Cell;

var dna = new DNA({
    membrane: {
        Server: {
            source: "membrane.Server"
        }
    },
    plasma: {
        Router: {
            source: "plasma.Router"
        },
        CSS: {
            source: "plasma.CSS",
            file: "./css/styles.css"
        },
        Render: {
            source: "plasma.Render",
            templates: "./tpl/"
        }
    }
});

var cell = new Cell(dna);
```

The above code is just a definition for the DNA and Cell initialization. You can see we've placed our Server in the membrane and the Router, CSS, and Render in the plasma, as we discussed in the last section. The `source` property is actually mandatory and contains the path to your individual organelles.

Keep in mind that the `file` property in the CSS organel and the `templates` property in the Render organel are actually custom properties, which I set. You can add whatever customization you need in here as well.

And just for your reference, the directory structure for your app should look like this:

```
/css
    /styles.css
/membrane
    /Server.js
/node_modules
/plasma
    /CSS.js
    /Render.js
    /Router.js
/tpl
```

A Basic Organel

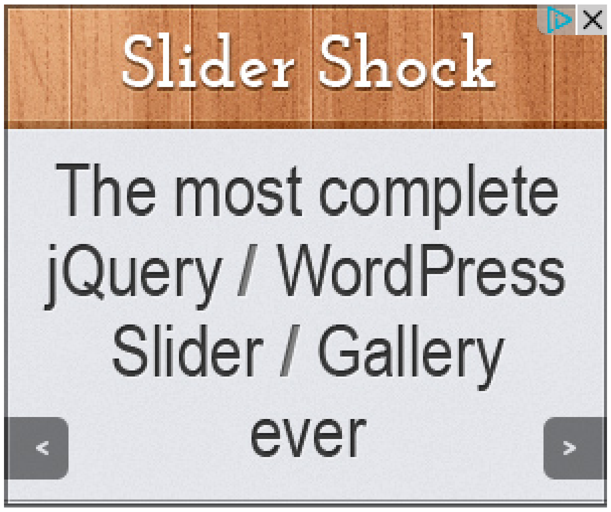
```
var Chemical = require("organic").Chemical;
var Organel = require("organic").Organel;
var util = require("util");

module.exports = function YourOrganelName(plasma, config) {
    Organel.call(this, plasma);
    // your custom logic here
}

util.inherits(module.exports, Organel);
```

The above code shows the basic format for creating an organel. If you want to use `this.emit` or `this.on` you'll need to make sure to inherit Organel as we did above. And actually, the `plasma` parameter variable has those exact same methods (`emit` and `on`), so you could use `plasma` directly and skip the inheritance if you wanted.

Also, notice the `config` parameter; This is the object that you defined in your DNA, which is a good place for any of your custom configuration.



The Server

The Server is your main organel, which accepts requests and sends responses to the browser. Here's how your Server organel should look:

```
var port = 3000;
module.exports = function Server(plasma, config) {
    Organel.call(this, plasma);

    var self = this;
    http.createServer(function(req, res) {
        console.log("request " + req.url);
        self.emit(new Chemical({
            type: "request",
            req: req
        }), function(html) {
            res.writeHead(200);
            res.end(html);
        });
    }).listen(port, '127.0.0.1');
    console.log('Server running at http://127.0.0.1:' + port + '/'

}
```

Two things are happening here. The first one is the definition of the NodeJS server, which of course has a handler accepting request (`req`) and response (`res`) objects. Once the request is received, the Server organel sends a chemical, with the type `request`, notifying the rest of the organelles. It also attaches the `req` object, so whoever needs more information about the incoming request can access data from the chemical directly.

The `emit` method then takes a second argument which is a callback function. You can use this to return the flow back to the organel, which sends the chemical. I.e. once the Render finishes its job, it calls the Server's callback. It takes the produced HTML and by using the `res` object sends the page to the user.

The Router

For our next organel, the Router just listens for a `request` chemical, which is sent by the Server. It gets the URL from the `req` object and decides which page should be shown. Here's the code for the Router:

```
module.exports = function Router(plasma, config) {
    Organel.call(this, plasma);

    var self = this;
    this.on("request", function(chemical, sender, callback) {
        var page = chemical.req.url.substr(1, chemical.req.url.len
        page = page == "" || page == "/" ? "home" : page;
        self.emit(new Chemical({
            type: "page",
            page: page,
            ready: callback
        }));
    });

}
```

Now, the router itself just emits a new chemical with a type of `page`. Keep in mind, there are two other organels listening for this chemical as well, but by default, it's not transfered to all of the other elements in the plasma. Of course, there may be times when you will need such functionality. To do so, you just need to `return false;` in the chemical's listener. We'll see this in action in the next section.

CSS Styles Provider

```
module.exports = function CSS(plasma, config) {
    Organel.call(this, plasma);

    var cssStyles = fs.readFileSync(config.file).toString();
    var self = this;
    this.on("page", function(chemical) {
        self.emit(new Chemical({
            type: "css",
            value: cssStyles
        }));
        return false;
    });

}
```

This module is just a simple one-task organel which gets the path to the `.css` file, reads it, and later emits a chemical containing the actual CSS styles. Also, pay attention to the `return false;` statement at the bottom. As I said from the last section, it's important to do this, otherwise the Render will not receive the `page` chemical sent by the Router. This happens because the CSS organel is defined before the Render in the DNA.

The Render

And lastly, here's the code for our Render organel:

```
module.exports = function Render(plasma, config) {
    Organel.call(this, plasma);

    var getTemplate = function(file, callback) {
        return fs.readFileSync(config.templates + file);
    }
    var formatTemplate = function(html, templateVars) {
        for(var name in templateVars) {
            html = html.replace("{" + name + "}", templateVars[nam
        }
        return html;
    }
    var templates = {
        layout: getTemplate("layout.html").toString(),
        home: getTemplate("home.html").toString(),
        about: getTemplate("about.html").toString(),
        notFound: getTemplate("notFound.html").toString()
    }
    var vars = {};
    var self = this;

    this.on("css", function(chemical) {
        vars.css = chemical.value;
    });
    this.on("page", function(chemical) {
        console.log("Opening " + chemical.page + " page.");
        var html = templates[chemical.page] ? templates[chemical.p
        html = formatTemplate(templates.layout, {content: html});
        html = formatTemplate(html, vars);
        chemical.ready(html);
    });

}
```

There are two helper methods here: `getTemplate` and `formatTemplate` which implement a simple template engine for loading an external HTML file and replacing mustache-style variables. All of the templates are stored in an object for quick access. Afterwards we have just a few lines for HTML formatting and then everything is ready to go. The Render organel also listens for the `css` chemical and lastly the application provides a `notFound` 404 page, if needed.

So here's what the final app's directory structure looks like:

```
/css
    /styles.css
/membrane
    /Server.js
/node_modules
/plasma
    /CSS.js
```

```
        /Render.js
        /Router.js
    /tpl
        /about.html
        /home.html
        /layout.html
        /notFound.html
```

Running the Application

Simply run `node index.js` in the console and you should see something similar to this:

With your server running, you should now be able to visit `http://127.0.0.1:3000` in your favorite browser. Try clicking on the links to switch between the two pages a few times and then go back to your console to view the output.

You should see a nice report about the applications recent activity. Now you may also notice something else in the console:

```
request /favicon.ico
Opening favicon.ico page.
```

You can see that there is one more request coming from the browser. It wants to load `favicon.ico`. However our little site doesn't have such an icon, so it just opens the `404` page. You can try this for yourself by visiting: `http://127.0.0.1:3000/favicon.ico`.

If you'd like to check out the full source code for this tutorial, you can download it using the download link at the top of this page.

Conclusion

In my opinion, Organic is a great concept. It's very flexible and encourages producing better applications. Keep in mind that the example in this article is based on my personal experience with other design patterns. So my use of terms like Router, Data Provider or Render is completely optional and you can change the names as you see fit. Feel free to experiment by creating new modules based on Organic and let me know what you think in the comments!

The core of Organic is developed by Boris Filipov and Valeri Bogdanov and I strongly recommend that you check them out on Github. If you are interested in using Organic, you will find things like Angel and WebCell really helpful as well.

Tags: node jsOrganic

By Krasimir Tsonev

Krasimir Tsonev is a coder with over ten years of experience in web development. With a strong focus on quality and usability, he is interested in delivering cutting edge applications. Currently, with the rise of the mobile development, Krasimir is enthusiastic to work on responsive applications targeted to various devices. Living and working in Bulgaria, he graduated at the Technical University of Varna with a bachelor and master degree in computer science. If you’d like to stay up to date on his activities, refer to his blog or follow him on Twitter.

Note: Want to add some source code? Type <pre><code> before it and </code></pre> after it. Find out more