# Development environment

## Front-end

The front-end is written in JavaScript (ES2015). We use a compiler that translates the ES2015 source code to ES5 code. It allows us to use more modern features of JavaScript without requiring browsers to support it (therefore supporting older browsers).

To start developing for the front-end, install Node.js (I use version 6.3.0, but any later version should work as well). Node.js comes with a package manager called ‘npm’, which we use to download our development tools (such as the JS compiler), as well as downloading the JS libraries that our application depends on. To download all the required dependencies, to into the client directory and type:

$ npm install

Node.js will now download all the required dependencies, and store them in the directory node\_modules/. This directory can be (should be) excluded from version control. Once it is done downloading the dependencies, you can start a development server:

$ npm start

A development server will be started on port 3000 (http://localhost:3000/). It will listen to changes made to the source files, and re-compile them automatically. So generally, there is no need to restart the server. You can use any editor to edit the files; there are some good editors available that are free to use, such as Atom and Visual Studio Code. I’ve used Atom, and if you wish to use it, I would suggest that you also install the ‘react’ package in Atom. (Go to the settings, packages tab, and find the ‘react’ package).

The project also contains a file called ‘package.json’. This is the file that contains the build configuration that is used by the JS compiler. From there, you can configure how the JS compiler should work, which dependencies are required, etc.

## Server

The server is built using the Play Framework (<https://www.playframework.com/>). The minimum required version is 2.5. Once you have installed the play framework (the ‘activator’), you can go into the envision-server directory start the activator console:

$ activator

It will also download the packages that are required for the server automatically. Once it is done with this, it will show a console. From this console, you can start the server:

[envision-server] $ run

It will by default start a server on port 9000. When you modify Java code, the activator will see this and automatically compile the code and restart the server when you access your server. There is no need to manually compile or restart the server (pretty much the same as the front-end development server). To stop the server, press Crtl-D.

It might be helpful to install an IDE to edit the Java code. I have used Eclipse (<http://www.eclipse.org/>). Installing Eclipse is just a matter of downloading one of the packages (I use the Java EE package), and double-clicking the Eclipse icon. Before you can open the server code in Eclipse, you must generate a project file from it. This can be done from within the activator. Type:

[envision-server] $ eclipse with-source=true

This will generate an Eclipse project file. The with-source=true is optional, as it will also download the sources for all the required dependencies. I highly recommend this, as this is very useful when you want to debug the application.

To debug the application, you have to start the activator with the ‘-jvm-debug’ parameter. For example:

$ activator -jvm-debug

This will start the activator, but the JVM will open port 9999 for remote debugging sessions. You can now start the server application (using the run command). In Eclipse, you can attach the debugger to the remote process. Go to the menu Run – Debug configurations. Select ‘Remote Java Application’, then click on the ‘New’ button in the toolbar. Select the project to debug (the envision server project), and set the connection properties (host: localhost, port: 9999). You can now debug the server application (e.g. set breakpoints by double clicking in the side line.)

The dependencies and configuration of the server are contained in the file ‘build.sbt’.

# Building a distribution

To build a distribution, two steps are required. First the front-end is built. Then the server application must be built. The build process of the server will then include the compiled front-end within the server distribution (the server application will serve the compiled JS/CSS/HTML of the front-end).

To build the front-end, go into the client directory, and type:

$ npm run build

This will compile the front-end. Then, go to the envision-server directory, and start the activator. Within the activator, type:

[envision-server] $ dist

This will build a zip file in the directory envision-server/target/universal. There is a ZIP file that can be unzipped. It will contain the compiled server application, including some scripts (in the bin/ directory) to start the server.

# Front-end architecture

The front-end is built using React (<https://facebook.github.io/react/>). The general idea is that there is one place in the application that maintains the state of the application. The view is entirely derived from this state, and the view code is therefore (mostly) stateless. To track changes to the state, we use MobX (<https://mobx.js.org/>). MobX tracks changes and will tell React which components need to re-render. React will then only update the relevant parts of the browser.

The nice thing about this, is that the way to write code for the view is entirely declarative. You just think about what you want to present on a screen for a given state, not how to get the view from the previous state to the next state. You could see the view as a pure function over the application state. Both React and MobX will take care of the most optimal way to update the DOM in the browser. See for example <https://rauchg.com/2015/pure-ui>.

The application state is defined in AppState.js. This file also contains additional functions to make updates to the state. The file commands.js contains most of the logic that is executed from the options in the menu, including all the required code to access the REST API of the envision server.

A react application is organized around the concept of components. The idea is that a component is a self-contained part of the application that can be reused. The components of our application can be found in the components/ directory. A component could also be seen as a simple function: given some input (application state), it will return the HTML that should be rendered for that application. The App.jsx component is the root component. Rendering this component will render the entire application. It will hand over rendering of subsequent parts to other components. Of particular interest are:

* Canvas – renders the canvas.
* Block – renders a block on the canvas.
* PostIt – renders a post it. We use the terms post-it and item (during the project, we discovered that the term post-it is patented, so we switched to the term item, but this is not yet subsequently updated in the code.)
* Menu – renders the menu.

The rest of the components are mostly dialog related. I believe they pretty much speak for themselves, but don’t hesitate to ask!

Most of the components will include embedded HTML mixed with JavaScript. This is a language called JSX. It can be used to output HTML (or SVG) directly from JavaScript. The ‘react’ package for Atom will understand this language, and the JS compiler also understands this language.

# Server architecture

The server application is fairly straight forward. The API is defined in the file conf/routes. It maps HTTP verbs and URI’s to Java code. The code is implemented in controllers/EnvisionController.java. For each request, it checks if the user is authenticated (with the Envision platform) and then queries the database.

# Authentication

To start the front-end, the server has to pass credentials to it. It will do this by settings cookies. This is implemented in EnvisionController.index() and EnvisionController.postIndex(), to support GET and POST requests respectively. The server will then serve the front-end files (JS/CSS/HTML) with the associated cookies. The client will pick this up and store this in its state. See the function authenticationInfo() in AppState.js.

During development, it might help to set fixed user credentials. Uncomment the lines:

//userId = userId || "<userId>";

//secret = secret || "<secret>";

Provide an existing userId and secret. Be sure to comment the code before building an application (the compiler will get rid of all the comments, so you don’t have to worry that these credentials appear in the compiled JS).

# Templates

The server will serve templates from the file public/templates.json. This is a JSON file with a list of template models. The easiest way to add templates is by using the export function in the front-end. By default it is hidden, but it can be shown by pretting and holding Alt + Shift while the menu is opened in the front-end. Simply copy/paste the entire JSON structure and add it to the templates.json file. Currently the file is an empty JSON array, new models can be added in this array (don’t forget to separate the various models with a comma).