BPjs Online IDE
Amir Dimri | Guy Shuster | Ofek Ashkenazi

Requirements Document

BPjs Online IDE

Chapter 1 - Introduction

1.1. The Problem Domain:

The domain of our project is the domain of Integrated Development Environments - IDE'S.

An IDE is a software application that provides comprehensive facilities to computer programmers for software development.
It normally includes facilities such as a source code editor, build automation tools (compiler, console etc.), and a debugger.
It includes traditional features such as syntax coloring, syntax highlighting, code search and code refactoring.
Also, most modern IDE's include more complex features such as version control, file and project browsing, test runners, coverage and more.
We will include the main IDE facilities and some of the traditional features in our project, on which we will expand later.

An online IDE may differ from a desktop IDE.
The main difference is that it can be accessed from a web browser, enabling software development on remote\cloud based devices rather than a private desktop computer.
Online IDE's do not usually contain all of the same features as a traditional or desktop IDE's, although all of the main facilities and basic features described above are included.

1.2. Context:

There are a few major components to our system, which will be implemented as a Client - Server architecture.

The client will be the web based IDE itself, with all included features, and the server will be the BPjs compiler and debugger.

1.3. Vision:

The main goal of our project is creating a comfortable platform for practicing and learning the BP paradigm, that can be used by both new and experienced BP developers.
An accessible environment for coding with some features that will make the learning and programming experiences easier and more effective for the users.

1.4. Stakeholders:

- **New BP developers**
- **Experienced BP developers**
- **BP researchers and inventors**

The main cycle is one where BP researchers update all changes and new features in the BP language, and the developers (who are the end users in our case) can experiment and learn the latest updates in BP.
This creates a very responsive supply and demand environment, beneficial to both creators of BP and developers.

1.5. Software Context:

Our system will consist of a window for writing code, a BP syntax toolbox for available BP expressions, control buttons (for example: Run, Debug), and a console for code output.
In addition, the code will be colored according to the BP and JavaScript syntax, and statically analysed as well.

The input of our system will be the code the user writes in the code window. He may run, debug it, save or load it, and even beautify the code at a push of a button.

The outputs of our system may vary.

They may be compilation errors, legal code outputs, or debugging results such as variable values for each b-thread,
or location of each b-thread in it's given code.
All of the outputs besides debugging info will be shown in the output console.
Debugging output info is still an open question, which will be decided soon.

Chapter 2 - Usage Scenarios

2.1. The Actors:

Although all the actors in our system are end users, they may have several different characteristics to them.
We predict that the largest percentage of users will consist of new BP developers, willing to get acquainted with the language in order to develop a system with it, or put it into practical use of some kind.
Other users may include experienced BP programmers willing to test a minor program while not downloading the full BP compiler onto their desktop computer (for convenience), or researchers willing to get familiar with the BP paradigm after hearing about it from colleagues in convenciones or reading about it in research papers.

2.2. Use Cases:

2.2.1.

Use Case Name: Editing code

Actors:

- System user

Trigger:

- The user puts the cursor on to the main code window or to the external event console

Preconditions:

- There is a network connection and the page was loaded properly

Postconditions:

- None

Normal Flow:

1. **The user opens the browser and enters the BPjsOnlineIDE website.**
2. **The user puts the cursor on to the main code window or to the external event console**
3. **The user can now write and edit the code as he wishes**

2.2.2.

Use Case Name: Executing Code

Actors:

- **System user**

Trigger: The user presses the “run code” button.

Preconditions:

- **The user wrote some code.**

Postconditions:

- **The code has been compiled.**

Normal Flow:

1. **The system user has written code that he wants to run, and maybe even added external events in the event console.**
2. **The system user runs the code he wrote.**
3. **The output of the program is shown in the output console.**

2.2.3.

Use Case Name: **Debugging code**

Actors:

- **System user**

Trigger:

- **The user presses the “debug code” button**

Preconditions:

- **The user wrote some code and set a breakpoint**

Postconditions:

- **The user is able to see the values of variables of different b-threads, global variables, states of b-threads, a trace of the program thus far and all current events divided into groups (requested, blocked, waiting etc.).**

Normal Flow:

1. **The system user types code to be executed in the appropriate place.**
2. **The system user sets one or more breakpoints, and presses the Debug button**
3. **The window switches to debug display (more on this in chapter 5)**
4. **The program stops at the first breakpoint and the user is able to see the data described above, and also debug his code with the following functions:**

   Step:
   Each press on the ‘next step’ button will make the code jump from the current bp.sync to the next one, without considering extra breakpoints.

Continue:
Each press on the 'next breakpoint' button will make the code jump from the current breakpoint to the next one (if present). If there is no breakpoint ahead, the current run will terminate and it's result will be displayed in the output window.

Terminate:
Finishes the current run of the program, with the output displayed in the output window.

Alternate Flow:

If the user doesn't set a breakpoint, the program will run until its end.

Chapter 3 - Functional Requirements

Capabilities and Functionalities of the System:

3.1. Compile and Run Code

In our system the user can write code and then run it (compilation is included). The code will be written in a text-box. This command is executed by a click of the "RUN" button.
This is the highest priority feature - without it there is no main functionality to the project.
Like mentioned earlier, our system will support Client-Server implementations.
The code will be sent to the server and compiled there upon a click of the "RUN" button.

3.2. Static Analysis

The code in our system will undergo a process of live static analysis using the Ace embeddable editor (more on this in chapter 5).
This means that the code will be checked for structural and syntactic errors in an "on the fly" manner using the Ace live syntax checker.
The system will alert the user of the type of error and place where it occurred by highlighting the problematic code, and showing the error type if the user hovers over that code using the mouse.
Only JavaScript with a BP extension will be considered correct syntax. Anything that will not align with that term, will be highlighted and considered a syntax error.
For example: Not closing block brackets, a function with no parentheses in it's declaration, error in the spelling of reserved JS words, using a variable without declaring it, case statements in a switch that don't have a break statement, comments within comments etc.
Not only will our system catch syntax errors, it will make optimizations by notifying unreachable code, statements that don't do anything, curly braces without an if, for, while, etc and regular expressions that are not preceded by a left parenthesis, assignment, colon, or comma.

3.3. Live Syntax Coloring And Highlighting

Our system will support live syntax coloring and highlighting. The user's code will be written in colors. Each color has a special meaning for the program. The system will highlight comments, variables, function names and all other key words in the BP language in different colors. We

suppose that this feature has no risks and can be done at this time frame. This feature makes the code easier to read and more understandable than without colors.

3.4. Debugger

In our system the user will be able to debug his code. The debugger will have some basic features:

a) **Run the debugger: start the running of the debugger on the user's program.**
b) **Breakpoint: the user can put a breakpoint in his program. A breakpoint is a line in the program that the user chooses, and the run stops there (the system executes the code until there).**
c) **Step Next: after executing the code until the breakpoint the user can execute his code a "sync" at a time - step by step execution.**
d) **Step Back: get back to the last step that the user did. Optional only after the step over command.**
e) **Trace: an option to look at all the events that happened during runtime in a chronological order. There will also be an option to click on one of the states that happened to go back to it (by state we mean all of the variables values and locations in the program).**
f) **Stop: finish the run with the debugger and terminate the program.**
g) **Var table: during debugging a table of variables of each b-thread and their current values are going to be shown to the user. This helps the user to inspect the code and find his mistakes.**

As far as UI goes, the user is able to see the values of variables of b-threads, states of b-threads, a trace of the program thus far and all current events divided into groups (blocked and not blocked).
In comparison to other functionalities that we have, this feature will be the hardest of them all to implement, and will take the most of the time - has a big risk.

3.5. Input - Output

In our system there will be a big text-box we call a "code editor" for writing the code.
In addition, the system will have a console, on which the outputs of the programs will be displayed. The console will also display errors that came up during the runtime of the program.
This is a "must have" feature (the part of showing the program outputs).

3.6. Refactoring

Our system will support a limited refactoring option - replace a word by another word:

a) Replace All - replace in all places which the word appears.
b) Replace - replace in a specific place.

This is mostly used with variable/function/class names for faster coding.

3.7. Find

In our system it will be possible to search for a word in the code (Ctrl+F). The user will be able to write a word and the system will search for this word in his .
This feature will work not only on full words, but on sub-words as well. For example: if the user searched for the word eve, all the "event" appearances in the code will be highlighted.

3.8. Download Code

The user will be able to save the current file on his computer and close the system. When the user starts the system and loads the file to the system, he will be able to continue from the point he stopped at.

3.9. File Loading

Loading files that were saved/written in advance.
The user will be able to load a text/js file continue his work from the point where he stopped.

3.10. Beautify

Arrange the indents in the code, by clicking a button. This feature can be important for beginners who are just learning to code and still don't pay attention to indentations, and also for advanced users who refactor the code a lot.

3.11. Theme

The ability to change the colors of the text editor and of the code.
To choose which color of text you want from some options.

3.12. Add external Event

The ability to add an event to the code while running.

3.13. Add sentence to the code editor

Ability to add a BPjs statement sentence to the code editor by clicking a button.

3.14. Stop button

Ability of the user to stop the run of the program on his code any time he wants.

Functional Requirements Summary

	Functionality	*Priority*	*Risks*	*Comments*
1	**Compile and Run Code**	**High**	**Depends in the architecture. Only client side is much easier than Client-Server. The reason is that the BPjs interpreter is simple to write, more than interfacing with the compiler on the server side.**	**Easy implementation and no scalability versus hard implementation but scalable and maintainable system. Depends on our time frame.**
2	**Static Analysis**	**Medium**	**Difficulty in covering the scope of all existing errors.**	**Mainly structural and syntactic errors.**
3	**Live Syntax Coloring And Highlighting**	**Medium**	**Almost none, should be simple.**	**-**
4	**Debugger**	**Medium**	**Hard to do, and possibly will take most of the time in our project.**	**BPjs is runs in a concurrent manner, and it will make the debugger more complicated. Also, there is not much information about building debuggers online.**
5	**Input-Output**	**High**	**-**	**A text-box for writing the code, and A console that will display the output of the code or errors that were found by the static analysis.**
6	**Refactoring**	**Low**	**-**	**Nice option, but not so necessary.**
7	**Find**	**Low**	**-**	**Nice option, but not so necessary.**

8	**Download Code**	**Low**	**-**	**Not a very important functionality. Our IDE is built for learning so saving files is not so necessary.**
9	**File Loading**	**Low**	**-**	**Necessary in case file saving exists.**
10	**Beautify**	**Low**	**-**	**Simple and convenient, yet not a necessary.**
11	**Theme**	**Low**	**-**	**Nice to have, not very necessary**
12	**Add External Event**	**High**	**-**	**Some programs must to get an event from the user.**
13	**Add sentence to the code editor**	**Medium**	**-**	**Not a very important functionality.**
14	**Stop button**	**High**	**-**	**important for infinity running.**

Chapter 4 - Non Functional Requirements

4.1. Implementation Constraints:

- **The UI (Frontend) will be implemented with TypeScript using Angular 6 and HTML5 in order to have a highly responsive environment, and full support from all browsers.**
  We will also include usage of Bootstrap and CSS for optimal design.
- **The Backend will implemented using Java's JAX-RS (specifically the Jersey library) to create a flexible REST API for the application.**
- **The text editor, JS static analyser and syntax highlighter with all it's features will all be included in an open source plugin embedded in our system (no need to reinvent the wheel).**
- **We will be using the Ace code editor for this task.**
- **Also, the system must be deployable to a remote server. This will be tested on a bgu server for proof of concept.**

4.2. UI Constraints:

The UI must meet the design requirements listed in the "Material Design" document by Google.
(https://material.io/design/guidelines-overview/#addition).
Google's guidelines will cover our requirements for the visibility and user experience of the system.

4.3. Special Restrictions & Limitations

The BPjs compiler that is given us needs to work.
It is treated as a black box.

Chapter 5 - Risk Assessment & Proof of Concept

In the next phase (the prototype) we wish to include limited functionality of the system described in the document above.
The first step we want to take in the project includes setting up the client-server architecture, and trying to run some BP code from the client's side on a demo server (localhost temporarily). After doing so, we will have a strong base upon which the rest of the project can be built.
Then, we will start designing the client side, trying to make the IDE as user friendly as possible, and including the syntactic features described in the previous sections.
Also, we will include the BPjs interpreter in order to make our system versatile as possible, and have a plan-b if the Client-Server plan won't work.
After searching the web for an open source javascript editor, finding options like highlight.js, Prism, SyntaxHighlighter and more, we found Ace.
Ace is an embeddable code editor written in JavaScript used by Cloud9, and it seems like the best option for our project.
It is easily embeddable, modifiable, and adding extra features to it (such as syntax highlighting for extra BP syntax to be added in the future) is easy and well documented. It supports all the features we wish to add to our project (and more).
We chose TypeScript as our main development language because we found Ace, and because we think that it suits the fact that BP is written in JS really well.
In conclusion, we wish to have a client-server architecture set up in our prototype,as well as a client-side only option, followed by the main grid of the client side IDE.
The prototype will also include some visual and syntactic features, all of which were described above, and that will be the next step in our project.