BPjs Online IDE
Amir Dimri | Guy Shuster | Ofek Ashkenazi

Maintenance Manual

BPjs Online IDE

In this document we will explain what each part of the system is responsible for, how to change / update parts of the existing system, and explain how to improve our system.

The system is built in client-server architecture.

Client:

The client consists of 3 layers:

1. **Presentation Layer:**
    - **Design - For design of our components we used Clarity and Angular Material.**
      https://clarity.design/
      https://material.angular.io/
    - **Components - The main component (App Component) contains several child components:**
        - **Header Component - This component contains all the main buttons for the system operation and the corresponding functions.**
          When switching to debug mode the buttons change accordingly.
          This component is very modular, and you can add buttons and functions as you wish to it.
        - **Code Editor Component - This component contains the code editor.**
          Changes can be made to the Ace code editor as detailed in the Ace Editor section.
        - **Output Component - This component contains the output of the program.**
          In transition to the debugger mode the component also contains global and local variable tables.
        - **Side Component - This component in normal mode contains three parts:**
            - **Adding an external event.**
            - **Adding regular sentences - sentences can be added to the *data.service.ts* file (array of sentence).**

- **Possible shortcuts in the code editor.**

In the debugger mode display, this component contains another three parts:

- **Adding external events.**
- **List of requested events when marked who is a blocked event and who is not blocked.**
- **Program trace.**

- **<u>Side Right Component</u> - This component is only shown in debug mode and contains a table of the existing bThreads in the system.**

Each component contains some information about the system and is a separate part of the other components.
The components receive most of the information they present from the BL layer.
Some components are affected by other components and a change in one component results to change in another component.

- **data.service.ts - This class contains all the common information of the components.**
  You can add information that can be used in several components.
  Any component that uses at least one of the data in this class creates an instance of this class in its constructor.

2. **BL: In This layer There are 4 Objects:**
   - **BreakPoint - which holds one field - 'line', with the type Number.**
   - **Debugger - manage all the stuff that is connected to the debugging operation. Holds:**
     - **<u>stepTrace</u> - array of the object DebugStep.**
     - **<u>eventTrace</u> - array of Events (type string).**
     - **<u>breakPoints</u> - array of the object breakPoints.**
     - **<u>stdout</u> - type: string, which contains the output that represent on the screen while debugging.**
     - **<u>programEnded</u> - type: boolean, true if the program is over.**
     - **<u>bpService</u> - type: BpService.**
   - **Debugger - manage all the stuff that connected to the debugging operation. Holds:**
     - **<u>bpService</u> - type: BpService.**
     - **<u>runner</u> - type: Runner.**
     - **<u>debugger</u> - type: Debugger.**
     - **<u>code</u> - type: string, which contain the user's code, that he wants to run or debug.**
   - **Runner - manage all the stuff that connected to the running user's code. Holds:**

- **bpService - type: BpService.**
- **isError - type: boolean, true if there was an error during running the user's code.**
- **stdout - type: string, which contains the output that represent on the screen while running.**
- **stop - type: boolean, true if the user click on the button 'stop' to stop the running on his code.**

3. **CL: In This layer There are 4 Objects:**
   - **WebSocketService - has one field webSocket - type: WebSocketSubject<any>.**
     This object communicates with the server, send the data to the server in 2 types of structure: Message or StepMessge. StepMessage structure is for sending the current step to the server (while debugging), for getting the next step, and the Message structure is for any other communicating with the server, for instance: init the user's code, run user's code, stop running, add external event.
   - **BpService - has one field connection - type: WebSocketService. For every operation which communicate with the server is needed, the connection (WebSocketService) send the appropriate type of message to the server.**
   - **BThreadInfo - The current state of a specific thread. It holds: The name of the thread, his local variables, firstLinePc, localShift which suppose to calc with them the current line that the thread reach. And also a boolean - isAdvance that it true if the thread is advanced in the user's program.**
   - **DebugStep - The current state of the user's program after the step. Holds:**
     - **bpss - an information for the server that indicate the state of the program. The client only save this information and don't use it.**
     - **map of vars - vals of the globals.**
     - **bThreads - array of BThreadInfo, which each one indicate the state of a specific thread in the user's program, and it locals.**
     - **reqList - request list.**
     - **selectableEvent.**
     - **waitList.**
     - **blockList.**
     - **selectedEvent - the current event that chosen to execute.**

- **line - the current line of the step, this line is equal to the current line of the thread that advanced from the existing bThreads. Each time only only one thread is advanced.**

Server:

In The server side, there are some objects:

1. **Server: When the system is open, it connects to the server by the function onOpen. When the client send an order to the server, it reach to the onMessage function. There, the server checks the type of the message, and by the type it calls for the wright command: init, run, step, add external event, or stop. This object holds the session with the client, and create a service for the client and save it.**
2. **Service: each client has a different service. when a request for comand reach the Server, it calls to the command in the Service. This object holds:**
   - **code - the user's code.**
   - **bprog - type: BProgram.**
   - **execService - type: ExecuteService.**
   - **runLogger - type: RunLogger, responsible for sending the events to the client while running.**
   - **rnr - type: BProgram Runner, run the user's code.**
3. **EncodeDecode: when a message reach to the server, it a json. With his object, we open the json and convert it to Message object or stepMessage object (depends in the type). And when we want to send a message back the client, we encode the message to string and send it.**
4. **RunLogger: send the events to the server while the server execution user's code.**
5. **SendBProgramRunnerListener: listener that while executing user's code, when it get an event it call to the runLogger to send it to the client.**
6. **Message, StepMessage, BThreadInfo - all those objects are exactly like the structure in the client, by this objects the client and the server communicate.**
7. **Step: This object execute the next step of user's code, while the user debugging. It holds: bpss - an information about the state of user's program, selecatableEvents, and selecetedEvent - the event that chosen.**

The Ace Editor

The editor injected in our project was upgraded with a few improvements of our own. The official documentation can be found here https://ace.c9.io/, and the improvements we added are detailed below.
PLEASE NOTE: Though the Ace editor is a remarkable tool, the Ace documentation is lacking and behind on many subjects.
There are not many tutorials online that talk about using Ace in a Typescript based environment such as Angular, so you might want to rely on documentation in the 'codeEditor' component as well.
We tried to note things that we discovered on our own after "looking for a needle in a haystack" for a while - both in the documentation and in the Ace source code itself.

- **Syntax coloring:**
  The method "setBpjsMode()" and the styles.css file is where all the magic happens.
  It is possible to add custom highlighting rules in that method, and add custom css styles to accompany these rules in the css file described above - All of this according to Ace documentation in https://ace.c9.io/#nav=higlighter.
- **Breakpoints:**
  We added custom breakpoints that can only be set on a line in the code that has "bp.sync" in it. This functionality is added in the "enableBreakpoints()" method.
  Also, some stylistic feature is added in the "enableMoveBreakpointsOnChange()" method, where a breakpoint moves with the original line it was set on if the code is edited (new line inserted for example).
  The style for the breakpoint and marker that comes with it is also in the styles.css file.
- **Debugger coloring features:**
  The debugging coloring feature we intended to add was coloring the line of the upcoming "bp.sync" in each bThread, but unfortunately we ran into some last minute issues and couldn't retrieve that line number from the BPjs server.
  Right now the code of the coloring is commented out because it paints random rows in the code, but the main coloring functionality is set and working. The only thing left to do is to plug in the correct line number for each bThread, and to change the marker styling to your liking in the styles.css file.

- **Adding syntax checking rules to Ace:**
  Currently, BPjs highlighting rules are just Javascript highlighting rules. Ace uses Web Workers to check the syntax, and we couldn't find any information online about how to extend the Javascript worker that Ace uses (especially because it uses a JSHint engine underneath, and not an extendable ESLint for example).

Possible Improvements to the Project

- **View: It's always possible to improve the view and make it more user friendly. Adding more common debugging features is always optional, and also improving existing view components.**
- **Line number: Right now, all the infrastructure is ready for getting a line number:**
  1. **Breakpoints: The user already has the option to mark lines with breakpoints in his program. When the server will return to the client the line number of each bThread, the system will be able to jump to the first breakpoint, and also make steps and steps back between breakpoints. All the functionality for those options is ready, and with a right line numbers it will work (right now, instead of returning the current line of each bThread the system returns the line of each bp.registerBThread (the field: firstLinePC by the function: getNextSyncLineNumber() in the class: BThreadInfo).**
  2. **Mark the current line of each bThread with the colors: green - for the thread that advanced from the last step, red - the others (threads that didn't advance). The function that marks the line of the thread thatget advanced in green is ready and commented in the component: CodeEditorComponent.**
  3. **BThreads table: in the bThreads table the system shows for each bThread its name and its current number. Right now, instead of returning the current line of each bThread the system returns the line of each bp.registerBThread (the field: firstLinePC by the function: getNextSyncLineNumber() in the class: BThreadInfo).**