

PEMANFAATAN SMARTPHONE SEBAGAI PENGENDALI PERMAINAN BERBASIS WEB

PRIAMBODO PANGESTU—2013730055

1 Data Skripsi

Pembimbing utama/tunggal: **Pascal Alfadian Nugroho**

Pembimbing pendamping: -

Kode Topik : **PAN4301**

Topik ini sudah dikerjakan selama : **1** semester

Pengambilan pertama kali topik ini pada : Semester **43** - Ganjil **17/18**

Pengambilan pertama kali topik ini di kuliah : **Skripsi 1**

Tipe Laporan : **B** - Dokumen untuk reviewer pada presentasi dan **review Skripsi 1**

2 Detail Perkembangan Penggerjaan Skripsi

Detail bagian pekerjaan skripsi sesuai dengan rencana kerja/laporan perkembangan terakhir :

1. Melakukan studi literatur mengenai *WebSockets*, *Socket.io*, *Node.js*, *Canvas API*, dan *Express.js*.
status : Ada sejak rencana kerja skripsi, dan ada penambahan studi literatur mengenai *Express.js* pada semester ini, serta perubahan tulisan *HTMLCanvas* pada rencana kerja menjadi *Canvas API*.
hasil :

(a) **Node.js**

Node.js adalah *JavaScript runtime* yang dibangun berdasarkan *V8* yang merupakan *JavaScript engine* milik perusahaan *Google* [1]. *Node.js* memiliki model *event-driven*, dan *non-blocking I/O* yang membuat teknologi tersebut efisien dalam implementasinya. Teknologi ini menyediakan beberapa kelas yang berfungsi untuk mengimplementasi fitur-fitur yang dimiliki.

Beberapa kelas yang terdapat pada *Node.js* yaitu sebagai berikut:

HTTP

Interfaces HTTP pada *Node.js* digunakan untuk menangani *request* dari protokol *HTTP* yang secara *native* sulit untuk digunakan. *Interface* ini akan menangani protokol *HTTP* dengan tidak melakukan *buffer* pada seluruh *request* atau *responses*.

Berikut akan dijelaskan kelas-kelas yang ada pada *interface HTTP*.

- **http.IncomingMessage**

Objek dari kelas ini akan dibuat oleh kelas *http.Server* atau *http.ClientRequest* dan memasukannya sebagai argumen suatu *event 'request'* dan *'response'*. Objek tersebut dapat digunakan untuk mengakses status *response*, *headers*, dan data. Kelas ini mengimplementasi *interface Readable Stream*, beserta *method*, *events*, dan properti yang ada didalamnya.

Properti:

- **message.headers**

Kembalian: *headers* milik objek *request/response*.

- **message.rawHeaders**

Kembalian: bentuk *raw* dari *headers* milik objek *request/response*.

- **message.statusCode**

Kembalian: tiga digit kode status *HTTP response*. Contoh: 404.

- **message.statusMessage**

Kembalian: pesan status *HTTP response* Contoh: *OK* atau *Internal Server Error*.

- **message.url**

Kembalian: *URL string* yang muncul pada permintaan *HTTP*.

- **http.ClientRequest**

Objek dari kelas ini dibuat dalam kelas ini sendiri dan dikembalikan dari *method http.request()*.

Objek ini merepresentasikan permintaan yang sedang berlangsung dimana *header* objek tersebut sudah berada dalam antrian. *Header* masih dapat diubah dengan menggunakan *setHeader(name, value)* dan *removeHeader(name)*. *Header* yang asli akan dikirim bersamaan dengan *chunk* pertama dari suatu data atau saat memanggil *request.end()*.

Beberapa **event** yang dimiliki oleh kelas ini yaitu sebagai berikut:

- **'connect'**

Dipancarkan saat *server* merespon kepada permintaan.

- **'response'**

Dipancarkan saat suatu respon diterima atas permintaan saat ini.

- **'timeout'**

Dipancarkan saat suatu *socket* telah mencapai batas waktu untuk tidak aktif.

Beberapa *method* yang dimiliki oleh kelas ini yaitu sebagai berikut:

- **request.end([data[,encoding]][[,callback]])**

Parameter:

- * **data**

tipe: **string** atau **Buffer**

Data yang akan dikirim.

- * **encoding**

tipe: **string**

Bersifat opsional dan akan bernilai *utf8* apabila tipe parameter *data* berupa *string*.

- * **callback**

tipe: **Function**

Fungsi callback

Method ini akan mengakhiri proses pengiriman permintaan.

- **request.getHeader(name)**

Parameter:

- * **name**

tipe: **string**

Nama dari *header* yang dibutuhkan.

Kembalian: suatu *string* yang sesuai dengan parameter.

Method ini akan membaca seluruh *header* dalam permintaan dan mengembalikan bagian yang sesuai dengan parameter. Berikut contoh implementasi dari *method* ini:

```
const contentType = request.getHeader('Content-Type');
```

- **request.removeHeader(name)**

Parameter:

- * **name**

tipe: **string**

Nama dari *header* yang dibutuhkan.

Method ini akan menghapus *header* yang sudah ada pada objek *header*. Berikut contoh implementasi dari *method* ini:

```
request.removeHeader('Content-Type');
```

- **request.setHeader(name, value)**

Parameter:

- * **name**

tipe: **string**

Nama dari *header* yang dibutuhkan.

- * **value**

tipe: **value**

Nilai yang akan dimasukan pada objek *header*

Method ini akan menetapkan suatu nilai kepada objek *header*. Berikut contoh implementasi *method* ini:

```
request.setHeader('Content-Type', 'application/json');
```

- **http.Server**

Kelas ini merupakan turunan dari *net.Server*. *Event* yang dimiliki kelas ini yaitu sebagai berikut:

- **'close'**

Dipancarkan apabila *server* sudah ditutup.

Beberapa properti yang dimiliki oleh kelas ini yaitu:

- **server.listening**

Mengembalikan *boolean* yang menandakan apakah *server* melakukan proses *listening* untuk suatu koneksi atau tidak.

- **server.maxHeadersCount**

Mengembalikan *number* yang menandakan batas maksimum suatu *headers* yang masuk.

Nilai default dari properti ini yaitu 2000.

- **server.timeout**

Mengembalikan *number* yang menandakan *timeout* dalam milidetik.

Beberapa *Method* yang dimiliki oleh kelas ini yaitu:

- **server.listen()**

Memulai *server* *HTTP* melakukan proses *listening* untuk suatu koneksi.

- **server.setTimeout([msecs][,callback])**

Parameter:

- * **msec** nilai *timeout* dalam milidetik. secara default bernilai 120000 (2 menit).

- * **callback** fungsi *callback*.

Kembalian: objek *server*.

Method ini menetapkan nilai *timeout* untuk *sockets* dan memancarkan *event* '*timeout*' pada objek *Server*.

- **server.close([callback])**

Parameter:

- * **callback** fungsi *callback*.

Method ini menghentikan *server* untuk menerima koneksi baru.

Beberapa *Method* yang dimiliki oleh *HTTP* yaitu sebagai berikut:

- **http.createServer([requestListener])**

Parameter:

- **requestListener** fungsi yang akan secara otomatis ditambahkan pada *event 'request'* milik kelas *http.Server*.

Kembalian: objek *http.Server*

Method ini akan membuat objek *http.Server* untuk menangani *request* dari *client* dan memberikan *response* kepada *client*. Fungsi yang diberikan pada *method* ini akan dipanggil satu kali setiap *request* dibuat kepada *server*.

- **http.request(options[,callback])**

Parameter:

- **options**

Dapat berupa *Object*, *string* atau *URL*. Berikut jenis-jenis *options* yang dapat menjadi parameter:

- * **protocol**

tipe: **string**

Protokol yang digunakan.

- * **host**

tipe: **string**

Nama domain atau alamat *IP* milik server.

- * **hostname**

tipe: **string**

Nama lain untuk *host*.

- * **port**

tipe: **number**

Port untuk *server*.

- * **path**

tipe: **string**

Path untuk permintaan.

- * **headers**

tipe: **Object**

Objek yang berisi permintaan *headers*.

- * **timeout**

tipe: **number**

Nomor yang menentukan *timeout* dari suatu *socket* dalam milidetik.

- **callback**

tipe: **Function**

Fungsi *callback*.

Kembalian: objek dari kelas *http.ClientRequest*.

Method ini digunakan untuk menangani permintaan *HTTP* pada *server*. Berikut contoh implementasi dari *method* ini:

```
const options = {
  hostname: 'www.google.com',
  port: 80,
  path: '/upload',
  method: 'POST',
  headers: {
```

```

    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': Buffer.byteLength(postData)
  }
};

const req = http.request(options, (res) => {
  console.log('STATUS: ${res.statusCode}');
  console.log('HEADERS: ${JSON.stringify(res.headers)}');
  res.setEncoding('utf8');
  res.on('data', (chunk) => {
    console.log('BODY: ${chunk}');
  });
  res.on('end', () => {
    console.log('No more data in response.');
  });
});

req.end();

```

Events

Node.js dibangun berdasarkan arsitektur *event-driven* dengan sifat *asynchronous*, dimana jenis-jenis objek tertentu akan memancarkan suatu *events* secara berkala dan akan memanggil objek *Function ("listeners")*.

Sebuah objek yang memancarkan *events* merupakan turunan dari kelas *EventEmitter*. Objek tersebut akan memanggil *method* *eventEmitter.on()* yang memungkinkan satu atau beberapa fungsi dapat ditangani dalam suatu *event* yang dipancarkan oleh objek saat ini.

Kelas *EventEmitter* dapat didefinisikan dengan memanggil modul *events* seperti berikut :

```
const EventEmitter = require('events');
```

Sebuah *EventEmitter* akan memancarkan '*event*' '*newListener*' pada saat *listeners* baru akan ditambahkan, dan '*removeListener*' akan dipancarkan saat *listeners* saat ini akan dihapus.

Berikut merupakan beberapa *method* yang dimiliki oleh kelas *EventEmitter*:

- **eventEmitter.on(eventName, listener)**

Parameter:

- *eventName*, nama dari suatu *event* yang akan dipancarkan.
- *listener*, suatu fungsi *callback* yang akan menangani *event* dari *eventName*.

Kembalian: referensi kepada *EventEmitter*.

Method ini berfungsi untuk mencatat suatu listener yang akan digunakan. Fungsi *listener* yang menjadi parameter *method* ini akan ditambahkan ke *index* terakhir dari *array of listeners* pada *eventName*. Tidak akan ada pengecekan apakah fungsi *listener* sudah dimasukan sebelumnya. Oleh karena itu, pemanggilan *eventName* dan *listener* secara berulang akan menyebabkan fungsi *listener* dimasukan kedalam *array* dan dipanggil secara berulang. *Method* ini juga akan mengembalikan *reference* kepada *EventEmitter*, sehingga pemanggilan dapat saling menyambung dengan pemanggilan lainnya.

Berikut merupakan contoh implementasi dari *method* ini:

```
const EventEmitter = require('events');
```

```
class MyEmmit extends EventEmitter {}

const myEmmit = new MyEmitter();

myEmmit.on('event', () => {
  console.log('suatu event telah terjadi');
});
```

- **eventEmitter.emit(eventName)**

Method ini berfungsi untuk memicu suatu *event* yang akan dipancarkan. **Parameter:**

- *eventName*, nama dari sebuah *event* yang akan dipancarkan.
- *...args*, argumen tambahan yang akan diberikan pada *eventName*.

Kembalian: *true* apabila *event* memiliki *listener*, *false* jika tidak.

Method ini akan memanggil masing-masing *listener* yang sudah dicatat oleh *eventEmitter.on()* dalam *array of listeners* secara sinkronis, dimana beberapa *listener* tersebut mengacu pada *eventName* yang sama. Argumen yang diterima dari parameter akan diberikan pada masing-masing *listener*.

Berikut merupakan contoh implementasi dari *method* ini:

```
const EventEmitter = require('events');
```

```
class MyEmmit extends EventEmitter {}
```

```
const myEmmit = new MyEmitter();
```

```
myEmmit.on('event', () => {
  console.log('suatu event telah terjadi');
});
```

Streams

Kelas ini digunakan untuk menangani aliran data yang terjadi pada *Node.js*. Data yang ditangani dapat berjumlah banyak dan akan menghabiskan banyak memori apabila tidak ditangani dengan baik. Oleh karena itu, modul *stream* menyediakan fitur-fitur yang memudahkan penanganan aliran data.

Ada empat tipe dasar *stream* dalam *Node.js*:

- **Readable**

Streams yang dapat membaca data dari sumber eksternal tertentu.

- **Writable**

Streams yang dapat menulis data dan mengirimkannya ke sumber eksternal tertentu.

- **Duplex**

Streams yang dapat membaca dan menulis data sekaligus.

- **Transform**

Duplex streams yang dapat memodifikasi atau mengubah data dimana data tersebut dapat dilihat langsung hasil perubahannya.

- i. **Readable Stream**

Merupakan abstraksi untuk sumber data yang digunakan. Berikut merupakan contoh dari *Readable Stream* pada *Node.js*:

- **HTTP responses** pada *client*
- **HTTP requests** pada *server*
- **fs read streams**

Seluruh *Readable streams* mengimplementasi *interface* yang didefinisikan oleh kelas *stream.Readable*.

- **stream.Readable**

Events:

- **'close'**

Event ini dipancarkan saat suatu *stream* atau sumber lain telah ditutup. *Event* ini menandakan tidak akan ada *event* lagi yang akan dipancarkan, dan tidak ada komputasi lain yang akan dilakukan.

```
const readable = getReadableStreamSomehow();
readable.on('close', (chunk) => {
  console.log('Stream telah ditutup');
});
```

- **'data'**

Event ini akan dipancarkan setiap kali suatu *stream* melepas kepemilikan sebuah data kepada pemakai. Hal tersebut dapat terjadi setiap suatu *stream* berganti menjadi mode *flowing* dengan memanggil *readable.pipe()*, *readable.resume()*, atau dengan menghubungkan *listener callback* pada *'data'* *event*. Contoh implementasi:

```
const readable = getReadableStreamSomehow();
readable.on('data', (chunk) => {
  console.log(`Menerima data sebesar ${chunk.length} bytes.`);
});
```

- **'end'**

Event ini dipancarkan saat tidak ada lagi data yang akan digunakan dari *stream*. Contoh implementasi:

```
const readable = getReadableStreamSomehow();
readable.on('end', () => {
  console.log('Tidak akan ada data yang dikirimkan.');
});
```

- **'error'**

Event ini akan dipancarkan oleh *Readable* setiap saat. *Event* ini dapat terjadi apabila *stream* tidak dapat menyediakan data dikarenakan kesalahan internal, atau ketika implementasi suatu *stream* mencoba mengirimkan *chunk* dari data yang tidak sesuai. Contoh implementasi:

```
const readable = getReadableStreamSomehow();
readable.on('error', () => {
  console.error('Terjadi kesalahan');
});
```

Method:

- **readable.pipe(destination[, options])**

Parameter:

*** destination**

tipe: *stream.Writable*

Destinasi untuk menulis suatu data.

*** options**

tipe: *Object*

Bersifat opsional, dapat berupa objek sebagai berikut:

end

tipe: *boolean*

Mengakhiri *writer* yang melakukan proses menulis saat *reader* telah selesai. Nilai *default* parameter ini yaitu *true*.

Kembalian: referensi yang menuju *destination*.

Method ini akan menghubungkan *Writable stream* pada objek *readable*, sehingga dapat berubah menjadi mode *flowing* secara otomatis dan akan menaruh seluruh data pada *Writable* yang sudah terhubung. *Method* ini mengembalikan referensi yang menuju *destination*, sehingga *pipe streams* dapat saling menyambung dengan *pipe streams* lainnya.

Berikut merupakan contoh implementasi dari *method* ini:

```
const fs = require('fs');

const readable = getReadableStreamSomehow();
const writable = fs.createWriteStream('file.txt');

// seluruh data dari objek readable masuk ke 'file.txt'
readable.pipe(writable);
```

(b) **Express.js**

Express.js merupakan *framework* aplikasi web untuk *Node.js* [2]. *Express.js* menyediakan fitur-fitur untuk web dan aplikasi *mobile* agar dapat bertahan lama. Untuk dapat menggunakan *Express.js*, dapat dilakukan langkah sebagai berikut:

```
var express = require('express');
var app = express();
```

Dengan begitu, fitur-fitur yang terdapat pada *Express.js* dapat digunakan untuk pengembangan aplikasi tertentu.

Berikut akan dijelaskan kelas-kelas yang terdapat pada *Express.js*:

express()

Untuk membuat aplikasi *Express*, langkah yang dilakukan adalah sebagai berikut:

```
const express = require('express');
const app = express();
```

Method yang dimiliki oleh fungsi *express()* yaitu sebagai berikut:

- **express.Router([options])**

Parameter:

- **options** bersifat opsional dan akan menentukan sifat dari objek *router*. Parameter ini dapat berupa beberapa jenis seperti berikut:
 - * *caseSensitive* memungkinkan *case-sensitive*. Dapat bernilai *true* atau *false*. Secara default akan bernilai *false*.

* *strict* memungkinkan *strict routing*. Dapat bernilai *true* atau *false*. Apabila bernilai *false*, maka parameter '/foo' dan '/foo/' akan dianggap sama oleh *router*.

Application

Kelas ini akan menangani berbagai proses yang terjadi dalam aplikasi *Express* seperti melakukan *routing* terhadap *HTTP requests*, mengatur *middleware*, *rendering* sebuah *HTML views*, dan mendaftarkan *template engine* tertentu. Untuk dapat melakukan fungsi-fungsi tersebut dapat dilakukan langkah berikut:

```
const express = require('express');
const app = express();
```

Baris pertama dari potongan kode tersebut berarti variabel *express* memanggil modul '*express*' agar dapat mengakses fungsi-fungsi yang ada pada modul tersebut. Sedangkan baris kedua, Objek *app* memanggil fungsi *express()* yang telah didapatkan dari variabel *express*.

Kelas ini memiliki beberapa *method* sebagai berikut:

- **app.all(path, callback[, callback ...])**

Parameter:

- **path** suatu *path* yang akan ditangani oleh *middleware*. Dapat berupa *string*, *path pattern*, atau *array* dari kombinasi *string* dan *path pattern*.
- **callback** merupakan fungsi *callback*, dimana fungsi tersebut dapat berupa fungsi *middleware*, kumpulan dari fungsi *middleware* (yang dipisahkan dengan menggunakan koma), fungsi *array of middleware*, atau kombinasi dari seluruh *item* tersebut.

Method ini dapat menangani seluruh *HTTP requests* seperti *GET*, *POST*, *PUT*, dan *DELETE*. Berikut merupakan contoh implementasi dari *method* ini:

```
app.all('/about', function(req, res, next){
  console.log('Mengakses bagian about ...');
  next(); //bagian ini akan menuju ke handler berikutnya
});
```

- **app.get(path, callback[, callback ...])**

Parameter:

- **path** suatu *path* yang akan ditangani oleh *middleware*. Dapat berupa *string*, *path pattern*, atau *array* dari kombinasi *string* dan *path pattern*.
- **callback** merupakan fungsi *callback*, dimana fungsi tersebut dapat berupa fungsi *middleware*, kumpulan dari fungsi *middleware* (yang dipisahkan dengan menggunakan koma), fungsi *array of middleware*, atau kombinasi dari seluruh *item* tersebut.

Method ini akan mengarahkan *HTTP GET requests* pada *path* dengan fungsi *callback* tertentu. Berikut merupakan contoh implementasi dari *method* ini:

```
app.get('/', function(req, res){
  res.send('Mengirimkan GET request pada homepage');
});
```

- **app.post(path, callback[, callback ...])**

Parameter:

- **path** suatu *path* yang akan ditangani oleh *middleware*. Dapat berupa *string*, *path pattern*, atau *array* dari kombinasi *string* dan *path pattern*.
- **callback** merupakan fungsi *callback*, dimana fungsi tersebut dapat berupa fungsi *middleware*, kumpulan dari fungsi *middleware* (yang dipisahkan dengan menggunakan koma), fungsi *array of middleware*, atau kombinasi dari seluruh *item* tersebut.

Method ini akan mengarahkan *HTTP POST requests* pada *path* dengan fungsi *callback* tertentu. Berikut merupakan contoh implementasi dari *method* ini:

```
app.post('/', function(req, res){
  res.send('Mengirimkan POST requests pada homepage');
});
```

- **app.route(path)**

Parameter:

- **path** suatu *path* yang akan ditangani oleh *middleware*. Dapat berupa *string*, *path pattern*, atau *array* dari kombinasi *string* dan *path pattern*.

Method ini akan mengembalikan instansi dari satu *route*, yang kemudian dapat digunakan untuk menangani *HTTP request* dengan *middleware* tertentu. Berikut merupakan contoh implementasi dari *method* ini:

```
app.route('/buku').get(function(req, res){
  res.send('Mendapatkan suatu buku');
});
```

- **app.use([path,] callback[, callback...])**

Parameter:

- **path** suatu *path* yang akan ditangani oleh *middleware*. Dapat berupa *string*, *path pattern*, atau *array* dari kombinasi *string* dan *path pattern*.
- **callback** merupakan fungsi *callback*, dimana fungsi tersebut dapat berupa fungsi *middleware*, kumpulan dari fungsi *middleware* (yang dipisahkan dengan menggunakan koma), fungsi *array of middleware*, atau kombinasi dari seluruh *item* tersebut.

Method ini akan menghubungkan *middleware* atau suatu fungsi tertentu dengan *path* yang sudah ditentukan. Dalam implementasi *method* ini, urutan penempatan pada baris kode sangat berpengaruh. Setelah *app.use()* dieksekusi, maka suatu *request* tidak akan mengeksekusi *middleware* yang ada dibawah baris kode *app.use()*. Berikut merupakan contoh implementasi dari *method* ini:

```
//request hanya akan sampai pada middleware ini
app.use(function(req, res){
  res.send('Hanya sampai sini saja');
});

//request tidak akan mengeksekusi baris ini
app.get('/', function(req, res){
  res.send('Hello World!');
});
```

- **app.listen(port, [hostname], [backlog], [callback])**

Parameter:

- **port** nomor yang akan dituju oleh server.
- **hostname** *string* yang diberikan pada gawai tertentu agar dapat dikenali. Parameter ini bersifat opsional.
- **backlog** nomor yang menentukan ukuran maksimal dalam antrian koneksi yang tertunda. Parameter ini bersifat opsional.

- **callback** merupakan fungsi *callback*, dimana fungsi tersebut dapat berupa fungsi *middleware*, kumpulan dari fungsi *middleware* (yang dipisahkan dengan menggunakan koma), fungsi *array of middleware*, atau kombinasi dari seluruh *item* tersebut. Parameter ini bersifat opsional.

Method ini akan menghubungkan suatu koneksi pada *host* dan *port* yang sudah ditentukan.

Berikut merupakan contoh implementasi dari *method* ini:

```
const express = require('express');
const app = express();
app.listen(3000);
```

Request

Sebuah objek dari kelas *Request* akan merepresentasikan *HTTP request* dan memiliki properti untuk *request query* seperti *body*, *HTTP headers* dan *parameters*.

Beberapa *method* yang ada pada kelas *Request* yaitu:

- **req.accepts(types)**

Berfungsi untuk memeriksa apakah tipe konten tertentu dapat diterima atau tidak.

- **req.get(field)**

Berfungsi untuk mengembalikan *HTTP request header* tertentu.

- **req.is(type)**

Berfungsi untuk mengembalikan apakah benar atau salah *type* pada parameter sama dengan status *Content-Type* pada *HTTP header*.

Response

Sebuah objek dari kelas *Response* akan merepresentasikan respon *HTTP* yang dikirim oleh *Express* pada saat menerima *HTTP request*.

Beberapa *method* yang terdapat pada kelas *Response* yaitu:

- **res.send([body])**

Parameter: *body* dapat berupa berbagai jenis objek seperti *Buffer*, *String*, dan *Array*.

Method ini akan mengirimkan respon *HTTP* kepada *client* sesuai dengan parameter yang diterima. Berikut merupakan contoh implementasi dari *method* ini:

```
// parameter objek String
res.send('Hello World!');
```

```
// parameter objek Array
res.send([1, 2, 3]);
```

```
// parameter objek Buffer
res.send(new Buffer('<p>This is a Buffer</p>'));
```

- **res.end([data][, encoding])**

Parameter:

- **data** dapat berupa objek *String* atau *Buffer* yang akan dikirim saat mengakhiri proses respon.
- **encoding** merubah suatu tipe data menjadi tipe data yang lain. Contoh beberapa tipe data yang tersedia yaitu *utf8*, *base64*, *ascii*, dan *hex*.

Method ini berfungsi untuk mengakhiri suatu proses respon. Apabila akan mengakhiri suatu respon tanpa memerlukan suatu data, maka dapat menggunakan *method* ini. Berikut merupakan contoh implementasi *method* ini:

```
app.get('/', function(req, res){
  res.end(); // apabila tidak memerlukan data.

  // apabila memerlukan suatu
  // data untuk mengakhiri proses.
  res.end('goodbye!');
});
```

- **res.render(view[, locals][, callback])**

Parameter:

- **view** suatu *string* yang menunjukkan *path* dari suatu *view file*.
- **locals** suatu objek yang memiliki properti yang menunjukkan variabel lokal dari *view*.
- **callback** suatu fungsi *callback*.

Method ini berfungsi untuk merubah *view file* dan mengirim *file* tersebut kepada *client*. Berikut merupakan contoh implementasi *method* ini:

```
app.get('/', function(req, res){

  // akan merubah(render) halaman about
  res.render('about');
});
```

- **res.sendStatus(statusCode)**

Parameter:

- **statusCode** kode status *HTTP*.

Method ini akan menetapkan kode status *HTTP* di parameter, dan akan mengirimkan bentuk *String* sebagai *body* dari respon. Berikut contoh implementasi *method* ini:

```
// akan mengirimkan 'OK'
// pada response body.
res.sendStatus(200);

// akan mengirimkan 'Not Found'
// pada response body.
res.sendStatus(404);

// akan mengirimkan 'Internal
// Server Error' pada response body.
res.sendStatus(500);
```

- **res.status(code)**

Parameter:

- **code** kode status *HTTP*.

Method ini akan menetapkan kode status *HTTP* untuk respon. Berikut merupakan contoh implementasi *method* ini:

```
res.status(403).end();
res.status(400).send('Bad Request');
```

- **res.json([body])**

Parameter:

- **body** dapat berupa tipe *JSON* apapun, seperti *array*, *String*, dan *Boolean*.

Method ini berfungsi untuk mengirimkan respon *JSON*. Berikut merupakan contoh implementasi *method* ini:

```
res.json({ user: 'tobi', age: '27'});
```

Router

Objek dari kelas *Router* merupakan *instance* dari *middleware* dan *routes*. Setiap aplikasi *Express* memiliki *router* secara *built-in*.

Method yang dimiliki oleh *Router* yaitu sebagai berikut:

- **router.METHOD(path, [callback, ...] callback)**

Parameter:

- **path** suatu *path* yang akan ditangani oleh *middleware*. Dapat berupa *string*, *path pattern*, atau *array* dari kombinasi *string* dan *path pattern*.
- **callback** merupakan fungsi *callback*, dimana fungsi tersebut dapat berupa fungsi *middleware*, kumpulan dari fungsi *middleware* (yang dipisahkan dengan menggunakan koma), fungsi *array of middleware*, atau kombinasi dari seluruh *item* tersebut.

Method ini menyediakan fungsionalitas *routing* dalam aplikasi *Express*, dimana *METHOD* merupakan salah satu *HTTP methods* seperti *GET*, *PUT*, dan *POST*, dalam huruf kecil. Dengan begitu, *method* ini dapat berupa *router.get()*, *router.post()*, dan *router.put()*.

Berikut merupakan contoh implementasi dari *method* ini:

```
// menggunakan HTTP method GET
router.get('/', function(req, res){
  res.send('hello world');
});

// menggunakan HTTP method POST
router.post('/buku', function(req, res){
  res.send('mendapatkan buku');
});
```

(c) WebSockets

WebSockets merupakan *Application Programming Interface (API)* yang memiliki kemampuan untuk membuka sesi komunikasi interaktif antara *browser* pengguna dan *server* [3]. Dengan *API* ini, pengguna dapat mengirim pesan ke *server* dan menerima respon tanpa harus melakukan *polling* pada *server* terlebih dahulu.

Berikut akan dijelaskan kelas-kelas yang ada pada *WebSockets*:

WebSocket

Kelas ini merupakan inti untuk mengakses fungsi yang ada pada *WebSockets*. Sebuah objek *WebSocket* dapat membuat dan mengelola koneksi *WebSocket* ke server, serta dapat mengirim dan menerima data pada koneksi tersebut.

Berikut merupakan konstruktor dari kelas *WebSocket*:

```
WebSocket WebSocket(in DOMString url, in optional DOMString protocols);
```

- **url**, parameter wajib yang menunjukan *URL* mana yang akan direspon oleh *WebSocket server*.

- **protocols**, parameter pilihan (tidak harus ada pada parameter) yang dapat berupa satu *string* atau *array of strings*. Parameter *protocols* merepresentasikan nama dari subprotokol yang akan digunakan oleh objek *WebSocket*. Apabila subprotokol tersedia pada parameter, maka *server* akan memeriksa apakah subprotokol tersebut dapat diterima atau tidak. *Server* akan memberikan respon apabila subprotokol dapat diterima, dan akan menghasilkan suatu *error* apabila tidak dapat diterima. Contoh subprotokol yang dapat digunakan yaitu:
 - **chat**
 - **superchat**

Konstruktor dari kelas *WebSocket* dapat menampilkan suatu *exception* seperti berikut:

SECURITY_ERR

Exception tersebut menandakan bahwa *port* yang akan digunakan untuk melakukan koneksi diblokir.

Atribut yang dimiliki oleh kelas *WebSocket* yaitu:

- **binaryType**
tipe: **DOMString**
Sebuah *string* yang menandakan tipe dari data biner yang dikirimkan oleh koneksi tertentu. Nilai dari atribut ini dapat berupa "*ArrayBuffer*" apabila objek dari *ArrayBuffer* digunakan.
- **bufferedAmount**
tipe: **unsigned long**
Jumlah *bytes* dari data yang belum dikirimkan oleh *method send()*. Nilai dari atribut ini akan kembali menjadi nol apabila seluruh data sudah dikirimkan. Apabila koneksi terputus, nilai atribut ini tidak akan kembali menjadi nol dan akan tetap bertambah apabila terus dilakukan pemanggilan pada *method send()*.
- **onclose**
tipe: **EventListener**
Event listener yang dipanggil saat atribut *readyState* dalam koneksi *WebSocket* berubah menjadi *CLOSED*. *Listener* akan menerima objek dari *CloseEvent* dengan nilai "*close*".
- **onerror**
tipe: **EventListener**
Event listener yang dipanggil saat terjadi *error*. *Event* tersebut akan bernilai "*error*".
- **onmessage**
tipe: **EventListener**
Event listener yang dipanggil saat atribut *readyState* dalam koneksi *WebSocket* berubah menjadi *OPEN*. Hal tersebut menandakan bahwa koneksi sudah siap untuk mengirim dan menerima data. *Event* tersebut akan bernilai "*open*".
- **protocol**
tipe: **DOMString**
String yang menandakan sebuah nama dari sub-protokol yang dipilih oleh *server*. Atribut ini akan menjadi salah satu masukan parameter yang dibutuhkan untuk konstruksi kelas *WebSocket*.
- **readyState**
tipe: **unsigned short**
Menunjukkan kondisi koneksi saat ini. Atribut ini memiliki beberapa konstanta yang menunjukkan kondisi dari koneksi *WebSocket*. Konstanta tersebut sebagai berikut:

– **CONNECTING**

nilai: 0

Koneksi belum terbuka.

– **OPEN**

nilai: 1

Koneksi sudah terbuka dan siap untuk melakukan komunikasi.

– **CLOSING**

nilai: 2

Koneksi sedang dalam proses menutup.

– **CLOSED**

nilai: 3

Koneksi sudah tertutup atau tidak dapat dibuka.

- **url**

tipe: **DOMString**

URL yang akan dituju oleh objek *WebSocket*. Atribut ini akan menjadi salah satu masukan parameter untuk konstruksi kelas *WebSocket*.

Kelas *WebSocket* memiliki dua buah *method*, yaitu:

- **void close(in optional unsigned long code, in optional DOMString reason)**

Berfungsi untuk menutup suatu koneksi atau menghentikan proses koneksi.

Parameter:

- **code** nilai numerik yang menunjukkan kode status, yang menjelaskan mengapa suatu koneksi ditutup. Apabila parameter ini tidak tersedia, maka akan diasumsikan dengan nilai *default* yaitu 1000 yang berarti transaksi selesai.
- **reason string** yang menjelaskan mengapa suatu koneksi ditutup.

Method ini dapat melemparkan eksepsi seperti berikut:

- **INVALID_ACCESS_ERR** parameter *code* yang tidak valid.
- **SYNTAX_ERR** parameter *reason* yang melebihi batas yang telah ditentukan.

- **void send(in DOMString data)**

Berfungsi untuk mengirimkan data ke *server* melalui koneksi *WebSocket*, dan menambah nilai dari *bufferedAmount* sebanyak jumlah *bytes* yang dibutuhkan untuk menampung data.

Parameter

Tipe data yang dikirimkan pada parameter dapat berbeda-beda, Beberapa tipe tersebut yaitu sebagai berikut:

- **USVString** sebuah teks *string* yang ditambahkan ke *buffer* dalam format *UTF-8*. Nilai dari *bufferedAmount* akan bertambah sesuai dengan jumlah *bytes* yang dibutuhkan untuk menyimpan *UTF-8 string*.
- **ArrayBuffer** data biner yang disimpan pada *fixed-length buffer*, dimana objek dari *ArrayBuffer* dimanipulasi oleh objek *TypedArray*.

Method ini dapat melemparkan eksepsi seperti berikut:

- **INVALID_STATE_ERR** koneksi saat ini tidak terbuka.
- **SYNTAX_ERR** parameter *data* tidak valid.

CloseEvent

Kelas ini akan menangani koneksi *WebSocket* yang ditutup. Objek *CloseEvent* akan dikirim ke *client* saat koneksi ditutup. Objek tersebut akan dikirimkan ke *listener* yang ditunjukan oleh atribut *onclose* milik objek *WebSocket*.

Konstruksi kelas ini yaitu:

- new **CloseEvent(typeArg, closeEventInit);**

Parameter:

- **typeArg**

tipe: **DOMString**

nama dari suatu *event* yang akan dikirimkan.

- **closeEventInit** bersifat pilihan, dan memiliki beberapa nilai sebagai berikut:

* *"wasClean"*

tipe: **boolean**

menunjukkan apakah koneksi sudah ditutup dengan baik atau belum.

* *"code"*

tipe: **unsigned short**

kode status yang menunjukkan mengapa koneksi ditutup.

* *"reason"*

tipe: **DOMString**

teks yang menunjukkan alasan mengapa koneksi ditutup oleh *server*.

Berikut merupakan nilai-nilai dari kode status koneksi ditutup:

- **0-999**

nama: -

Reserved. Tidak digunakan.

- **1000**

nama: **Normal Closure**

Penutupan normal, yang berarti koneksi sudah menyelesaikan apapun tujuan dari koneksi tersebut.

- **1001**

nama: **Going Away**

Endpoint menghilang karena kesalahan server atau *browser* tidak lagi mengakses halaman yang sudah membuka koneksi.

- **1002**

nama: **Protocol Error**

Endpoint menghentikan koneksi karena adanya kesalahan protokol.

- **1003**

nama: **Unsupported Data**

Koneksi dihentikan karena *endpoint* menerima data dengan tipe yang tidak bisa diterima (contoh: *text-only endpoint* menerima data biner).

- **1004**

nama: -

Reserved. Makna dari kode tersebut akan dijelaskan di waktu yang akan datang.

- **1005**

nama: **No Status Recieved**

Reserved. Menandakan bahwa tidak ada kode status yang tersedia.

- **1006**

nama: **Abnormal Closure**

Reserved. Menandakan bahwa koneksi ditutup secara tidak normal (contoh: tidak ada *close frame* yang dikirimkan).

- **1007**

nama: **Invalid frame payload data**

Endpoint menghentikan koneksi karena pesan yang diterima berisi data yang tidak konsisten (contoh: data *non-UTF-8* berada di dalam pesan teks).

- **1008**

nama: **Policy Violation**

Endpoint menghentikan koneksi karena menerima pesan yang melanggar kebijakan. Kode status ini dapat digunakan apabila tidak ada kode status lain yang cocok atau digunakan untuk tidak menunjukkan kebijakan lebih rinci.

- **1009**

nama: **Message too big**

Endpoint menghentikan koneksi karena menerima *frame* data yang terlalu besar.

- **1010**

nama: **Missing Extension**

Client menghentikan koneksi karena *server* tidak menangani satu atau beberapa ekstensi yang diminta oleh *client*.

- **1011**

nama: **Internal Error**

Server menghentikan koneksi karena mengalami kondisi tertentu yang menyebabkan tidak bisa memenuhi permintaan *client*.

- **1012**

nama: **Service Restart**

Server menghentikan koneksi karena harus mengulang kembali koneksi.

- **1013**

nama: **Try Again Later**

Server menghentikan koneksi karena ada kondisi yang harus ditangani untuk sementara (contoh: *overloaded*).

- **1014**

nama: **Bad Gateway**

Server bertindak sebagai *gateway* atau *proxy* dan menerima respon yang tidak benar dari *upstream server*.

- **1015**

nama: **TLS Handshake**

Reserved. Menandakan bahwa koneksi ditutup karena gagal melakukan *TLS handshake* (contoh: sertifikat *server* tidak dapat diverifikasi).

- **1016-1999**

nama: -

Reserved. Akan digunakan oleh standar *WebSocket* di waktu yang akan datang.

- **2000-2999**

nama: -

Reserved. Akan digunakan oleh ekstensi *WebSocket*.

- **3000-3999**

nama: -

Tersedia untuk digunakan oleh *libraries* dan *frameworks*.

- **4000-4999**

nama: -

Tersedia untuk digunakan oleh aplikasi.

Kelas ini merepresentasikan pesan yang diterima oleh suatu objek tujuan. *Constructor* dari kelas ini yaitu:

- **new MessageEvent(type, init);**

Parameter:

- **type**

Tipe pesan *MessageEvent* yang akan dibuat.

- **init**

Parameter ini dapat berupa beberapa nilai seperti berikut:

- * **data**

Data yang akan diisi pada *MessageEvent*. Dapat bernilai tipe data apapun.

- * **origin**

Merepresentasikan *origin* dari suatu pemancar pesan.

- * **ports**

Sebuah *array of MessagePort* yang merepresentasikan *port* yang berhubungan dengan saluran pesan yang sedang dikirim.

Contoh implementasi:

```
var myMessage = new MessageEvent('worker', {
  data : 'hello'
});
```

Beberapa properti yang dimiliki oleh kelas ini yaitu:

- **MessageEvent.data**

Merepresentasikan data yang dikirim oleh pemancar pesan. Parameter ini dapat berisi data dengan tipe data apapun.

Contoh implementasi:

```
myWorker.onmessage = function(e) {
  result.textContent = e.data;
  console.log('Message received from worker');
};
```

- **MessageEvent.source**

Merepresentasikan pemancar pesan atau sumber suatu pesan berasal.

Contoh implementasi:

```
myWorker.onmessage = function(e) {
  result.textContent = e.data;
  console.log('Message received from worker');
  console.log(e.source);
};
```

(d) Socket.io

Socket.io merupakan salah satu teknologi yang memanfaatkan protokol *WebSockets* [4]. Teknologi ini memungkinkan sebuah aplikasi untuk melakukan komunikasi dua arah secara *real-time*. *Socket.io* dapat dijalankan di setiap *platform*, *browser*, dan gawai.

Sebelum dapat menggunakan *socket.io*, *Node.js* harus sudah terinstall pada sistem komputer. Apabila hal tersebut sudah dilakukan, maka *socket.io* dapat diinstall dengan menggunakan *command line tools* atau sejenisnya dengan melakukan langkah seperti berikut:

```
npm install socket.io
```

Dengan begitu, aplikasi yang dibuat sudah dapat mengakses fitur-fitur yang dimiliki oleh *socket.io*. *Socket.io* dibagi menjadi dua *API*, yaitu *Server API* dan *Client API*. Berikut akan dijelaskan kelas-kelas yang dimiliki *Socket.io*:

SERVER API

Kelas-kelas yang ada pada *Server API* digunakan untuk menangani proses yang terjadi dalam *server*[5]. Kelas-kelas tersebut adalah sebagai berikut:

i. Server

Kelas ini merupakan inti untuk dapat menangani proses yang terjadi dalam *socket.io server*.

Kelas ini memiliki tiga konstruktor seperti berikut:

- **new Server(*httpServer*[, *options*])**

Parameter:

- **httpServer**

tipe: **http.Server**

Server yang akan dituju.

- **options**

tipe: **Object**

Parameter ini dapat berupa berbagai jenis objek. Objek-objek tersebut yaitu sebagai berikut:

- * **path**

tipe: **String**

Nama dari path yang akan ditangkap oleh *server* (contoh: `/socket.io`).

- * **serveClient**

tipe: **Boolean**

Menunjukan apakah *server* akan melayani *file* dari *client* atau tidak.

- * **adapter**

tipe: **Adapter**

Objek yang akan mengatur beberapa *socket* untuk menerima koneksi, dan mengirimkan pesan antara satu *socket* dengan *socket* lainnya.

- * **origins**

tipe: **String**

Origins yang diperbolehkan oleh *server*.

Untuk dapat menggunakan fitur yang ada pada *socket.io*, harus menambahkan modul *socket.io* pada konstanta tertentu. Hal tersebut dapat dilakukan dengan dua cara, yaitu menggunakan kata kunci *new* atau tanpa menggunakan kata kunci *new*:

- Menggunakan *new*

```
const Server = require('socket.io');
const io = new Server();
```

- Tanpa menggunakan *new*

```
const io = require('socket.io')();
```

Contoh implementasi konstruktor:

```
const Server = require('socket.io');
const http = require('http').createServer();
```

```
const io = new Server(http, {
  path: '/test',
  serveClient: false
});
```

- **new Server(port[,options])**

Parameter:

– **port**

tipe: **Number**

Nomor *port* yang akan dituju.

– **options**

tipe: **Object**

Sama seperti konstruktor pertama, parameter ini dapat berupa berbagai jenis objek.

Contoh implementasi konstruktor:

```
const Server = require('socket.io');
const io = new Server(3000, {
  path: '/test',
  serveClient: false
});
```

- **new Server(options)**

Parameter:

– **options**

tipe: **Object**

Sama seperti konstruktor pertama, parameter ini dapat berupa berbagai jenis objek.

Contoh implementasi konstruktor:

```
const Server = require('socket.io');
const io = new Server({
  path: '/test',
  serveClient: false
});
```

Beberapa *method* yang dimiliki oleh kelas ini yaitu sebagai berikut:

- **server.serveClient([value])**

Parameter:

– **value**

tipe: **Boolean**

Kembalian: *Server* atau *Boolean*.

Apabila parameter *value* bernilai *true*, maka *server* akan menangani *file* dari *client*. Apabila tidak ada argumen pada *method* ini, maka kembalian akan berupa status *default* dari *serveClient* saat ini (*true*).

- **server.path([value])**

Parameter:

– **value**

tipe: **String**

Kembalian: *Server* atau *String*

Parameter *value* akan menetapkan nilai dari *path* yang akan dituju. Secara *default* nilai dari *path* akan diisi dengan */socket.io*. Apabila tidak ada argumen pada *method* ini, maka kembalian akan berupa nilai dari *value* saat ini.

Berikut contoh implementasi dari *method* ini:

```
const io = require('socket.io')();
io.path('/myownpath');
```

- **server.adapter([value])**

Parameter:

– **value**

tipe: **Adapter**

objek *Adapter* yang akan digunakan.

Method ini akan menentukan *adapter* apa yang akan digunakan. Secara *default* adapter yang akan digunakan merupakan objek *adapter* yang berasal dari *socket.io* yang bekerja berdasarkan memori. Apabila *method* ini tidak menerima parameter, maka kembalian akan berupa *adapter* saat ini (secara *default*).

Berikut contoh implementasi dari *method* ini:

```
const io = require('socket.io')(3000);
const redis = require('socket.io-redis');
io.adapter(redis({ host: 'localhost', port: 6379}));
```

- **server.origins([value])**

Parameter:

– **value**

tipe: **String**

Menunjukkan *origin* mana yang diizinkan oleh *server*.

Kembalian: *Server* atau *String*

Method ini akan menetapkan *origins* mana yang diizinkan oleh *server*. Secara *default*, *origins* yang diizinkan dapat dari mana saja.

Berikut contoh implementasi dari *method* ini:

```
const io = require('socket.io')();
io.origins(['foo.example.com:443']);
```

- **server.attach(httpServer[, options])**

Parameter:

– **httpServer**

tipe: **http.Server**

Server yang akan dihubungkan.

– **options**

tipe: **Object**

Sama seperti konstruktor *new Server(httpServer[, options])*, parameter ini dapat berupa berbagai jenis objek.

Method ini akan menghubungkan *Server* dengan objek dari *engine.io* pada parameter *httpServer* dengan diberikan suatu *options*.

- **server.of(nsp)**

Berfungsi untuk menginisialisasi dan menerima *namespace* yang didapat dari tanda pe-

ngenal *nsp*.

Parameter:

- **nsp**
tipe: **String**
Namespace.

Contoh implementasi:

```
const adminNamespace = io.of('/admin');
```

• **server.bind(engine)**

Parameter:

- **engine**
tipe: **engine.Server**

Kembalian: *Server*

Method ini akan menghubungkan *server* dengan objek *Server* dari *engine.io*.

• **server.close([callback])**

Method ini akan menutup koneksi *server socket.io*. Parameter *callback* bersifat opsional dan akan dipanggil saat semua koneksi sudah ditutup.

Parameter:

- **callback**
tipe: **Function**
Fungsi *callback*.

Contoh implementasi:

```
const Server = require('socket.io');
const PORT = 3030;
const server = require('http').Server();

const io = Server(PORT);

// menutup server saat ini
io.close();
```

ii. Namespace

Kelas ini merepresentasikan kumpulan *sockets* yang terhubung dalam lingkup yang diidentifikasi oleh nama *path*. *Client* akan selalu terhubung ke / (*namespace* utama), kemudian dapat terhubung ke *namespace* lain saat berada dalam koneksi yang sama.

Beberapa properti yang dimiliki oleh kelas ini yaitu sebagai berikut:

• **namespace.name**

tipe: **string**
Nama dari *namespace* tertentu.

• **namespace.adapter**

tipe: **string**
Adapter yang digunakan untuk *namespace* tertentu.

Beberapa *method* yang dimiliki oleh kelas ini yaitu:

• **namespace.emit(eventName[, ...args])**

Berfungsi untuk memancarkan suatu *event* pada seluruh *clients* yang terhubung.

Parameter:

- **eventName**

tipe: **String**

Nama dari *event*.

- **args**

Argumen tambahan.

Contoh implementasi:

```
const io = require('socket.io')();
```

```
// akan memancarkan event pada namespace utama (/)
io.emit('an event sent to all connected clients');
```

- **namespace.to(room)**

Berfungsi untuk memancarkan *event* kepada *client* yang sudah bergabung dalam *room* tertentu.

Parameter:

- **room**

tipe: **String**

Nama dari *room*.

Kembalian: *namespace*.

Contoh implementasi:

```
const io = require('socket.io')();
const adminNamespace = io.of('/admin');
```

```
adminNamespace.to('level1').emit('an event',
{ some: 'data' });
```

- **namespace.clients(callback)**

Berfungsi untuk mendapatkan daftar *ID clients* yang terhubung pada *namespace* ini.

Parameter:

- **callback**

fungsi *callback*

Contoh implementasi:

```
const io = require('socket.io')();
io.of('/chat').clients((error, clients) => {
if (error) throw error;
```

```
// akan menampilkan id seperti [PZDoMHjiu8PYfRiKAAAF,
// Anw2LatarvGVVXEIAAAD]
console.log(clients);
});
```

iii. **Socket**

Kelas ini merupakan kelas yang mendasar untuk berinteraksi dengan *browser* milik *clients*. *Socket* merupakan milik *namespace* tertentu dan menggunakan kelas *Client* untuk berkomunikasi. Dalam setiap *namespace*, dapat ditentukan suatu *room* yang dimana sebuah *socket* dapat bergabung atau keluar. Kelas ini pun merupakan turunan dari *EventEmitter* milik *Node.js*. Kelas ini melakukan *override* pada *method* *emit* milik *EventEmitter*, dan tidak memodifikasi *method* lain.

Beberapa properti yang dimiliki oleh kelas ini yaitu sebagai berikut:

- **socket.id**

Tanda pengenal unik untuk sesi saat ini, yang didapatkan dari kelas *Client*.

- **socket.rooms**

Objek yang menandakan *room* dari *client* saat ini.

Beberapa *method* yang dimiliki oleh kelas ini yaitu sebagai berikut:

- **socket.join(room[, callback])**

Berfungsi untuk menambah *client* ke *room*.

Parameter:

- **room**

tipe: **String**

Nama *room*.

- **callback**

tipe: **Function**

Fungsi *callback*.

Kembalian: *Socket*.

Contoh implementasi:

```
io.on('connection', (socket) => {
  socket.join('room 237', () => {
    let rooms = Objects.keys(socket.rooms);
    console.log(rooms); // [ <socket.id>, 'room 237' ]

    //akan memancarkan kepada seluruh user
    //yang berada di room yang sama
    io.to('room 237', 'a new user
has joined the room');
  });
});
```

- **socket.leave(room[, callback])**

Berfungsi untuk menghapus *client* dari suatu *room*.

Parameter:

- **room**

tipe: **String**

Nama *room*.

- **callback**

tipe: **Function**

Fungsi *callback*.

iv. Client

Kelas ini merepresentasikan koneksi *engine.io* yang masuk. Kelas ini dapat berhubungan dengan banyak *socket* yang dimiliki oleh *namespace* berbeda.

Properti yang dimiliki oleh kelas ini yaitu sebagai berikut:

- **client.conn**

Merepresentasikan koneksi *engine.io* yang masuk.

- **client.request**

Berfungsi untuk mendapatkan permintaan *headers* seperti *Cookie* atau *User-Agent*.

CLIENT API

Client API digunakan untuk menangani proses pengaturan koneksi yang terjadi pada bagian *client*.

Kelas-kelas yang ada pada *Client API* yaitu sebagai berikut:

- i. **IO** Untuk dapat menggunakan fungsi yang ada pada *IO*, dapat dilakukan langkah seperti berikut:

```
// berfungsi untuk melayani file client
<script src="/socket.io/socket.io.js"></script>

<script>
const socket = io('http://localhost');
</script>
```

Dengan langkah tersebut, *socket.io* akan dapat menangani *file* yang berasal dari *client*. Selain langkah tersebut, ada satu langkah lagi yang dapat digunakan, yaitu sebagai berikut:

```
const io = require('socket.io-client');
```

Method yang dimiliki oleh kelas ini yaitu sebagai berikut:

- **io([url][, options])**

Berfungsi untuk membuat objek baru dari kelas *Manager* dengan *url*, dan akan menggunakan objek kelas *Manager* yang sudah ada untuk pemanggilan selanjutnya, apabila *multiplex* pada parameter *option* bernilai *true*. Objek *Socket* akan dikembalikan untuk *namespace* yang sudah ditentukan oleh nama *path* pada *URL*, dengan nilai *default* /.

Parameter:

– **url**

tipe: **String**

Nama *URL*.

– **options**

tipe: **Object**

Parameter ini dapat berupa beberapa jenis objek, seperti milik kelas *Manager*

Kembalian: *Socket*

- ii. **Manager** Konstruksi kelas ini yaitu sebagai berikut:

- **new Manager(url[, options])**

– **url**

tipe: **String**

Nama *URL*

– **options**

tipe: **Object**

Parameter ini dapat berupa berbagai jenis objek seperti berikut:

* **path**

tipe: **String**

Nama *path* yang dituju pada bagian *server*.

* **reconnection**

tipe: **Boolean**

Menandakan apakah akan melakukan koneksi ulang secara otomatis atau tidak.

* **timeout**

tipe: **Number**

Menandakan waktu koneksi yang habis sebelum *event connect_error* dan *connect_timeout* terjadi.

Beberapa *method* yang ada pada kelas ini yaitu sebagai berikut:

- **manager.timeout([value])**

Berfungsi untuk menentukan nilai *timeout* untuk suatu koneksi.

Parameter:

– **value**

tipe: **Number**

Nilai *timeout*.

Kembalian: *Number*

- **manager.open([callback])**

Apabila objek *Manager* diinisiasi dengan nilai *false* pada *autoConnect*, maka dapat menggunakan *method* ini untuk membuat percobaan koneksi baru.

Parameter:

– **callback**

tipe: **Function**

Fungsi *callback*.

Beberapa *events* yang ada pada kelas ini yaitu sebagai berikut:

- **connect_error**

Akan dipancarkan apabila ada kesalahan pada koneksi.

- **connect_timeout**

Akan dipancarkan apabila waktu koneksi telah habis.

iii. **Socket** Beberapa *method* yang dimiliki oleh kelas ini yaitu:

- **socket.open()**

Berfungsi untuk membuka *socket* secara manual.

Kembalian: *Socket* Contoh implementasi:

```
const socket = io({
  autoConnect: false
});
```

```
// ...
```

```
socket.open();
```

- **socket.emit(eventName[, ..args][, ack])**

Berfungsi untuk memancarkan *event* kepada *socket* yang ditandai dengan nama dari *event* tersebut.

Parameter:

– **eventName**

tipe: **String**

Nama *event*.

– **args**

Argumen tambahan (opsional).

– **ack**

tipe: **Function**

Fungsi tambahan (opsional).

Contoh implementasi:

```
socket.emit('ferret', 'tobi', (data) => {
  console.log(data);
});
```

- **socket.close()**

Berfungsi untuk menutup *socket* secara manual.

Beberapa *events* yang ada pada kelas ini yaitu sebagai berikut:

- **connect_error**

Akan dipancarkan apabila ada kesalahan pada koneksi.

- **connect_timeout**

Akan dipancarkan apabila waktu koneksi telah habis.

(e) Canvas API

Canvas API merupakan salah satu elemen *HTML5* yang digunakan untuk membuat gambar grafis dalam aplikasi web [6]. Teknologi ini memiliki fitur untuk membuat komposisi foto, membuat animasi, dan membuat *real-time video processing* atau *rendering*. Untuk dapat menggunakan fitur-fitur yang ada pada *Canvas API*, langkah yang harus dilakukan adalah sebagai berikut:

- i. Menambahkan tag `<canvas>` pada file *HTML*, dan menambahkan *id* yang akan digunakan pada file *JavaScript*.

```
<canvas id="canvas"></canvas>
```

- ii. Membuat variabel untuk mendapatkan konteks *rendering* dan fungsi-fungsi menggambar agar dapat menampilkan sesuatu pada `<canvas>`.

```
// Variabel yang akan menampilkan sesuatu
// pada <canvas> dengan id='canvas'
var canvas = document.getElementById('canvas');
```

```
// Variabel yang akan mendapatkan fungsi-fungsi menggambar
var ctx = canvas.getContext('2d');
```

Berikut akan dijelaskan beberapa *interface* dari *Canvas*:

HTMLCanvasElement

Interface ini menyediakan beberapa properti dan *method* untuk memanipulasi tata letak dan tampilan dari elemen *canvas*.

Beberapa properti yang dimiliki oleh *HTMLCanvasElement* yaitu :

- **HTMLCanvasElement.height**

Merupakan bilangan integer positif yang merepresentasikan tinggi dari atribut *HTML* pada elemen *canvas* yang diinterpretasikan dalam piksel *CSS*. Apabila atribut tidak didefinisikan, atau atribut diisi dengan nilai negatif, maka akan digunakan nilai *default* yaitu 150.

- **HTMLCanvasElement.width**

Merupakan bilangan integer positif yang merepresentasikan lebar dari atribut *HTML* pada elemen *canvas* yang diinterpretasikan dalam piksel *CSS*. Apabila atribut tidak didefinisikan, atau atribut diisi dengan nilai negatif, maka akan digunakan nilai *default* yaitu 300.

Beberapa *method* yang dimiliki oleh *HTMLCanvasElement* yaitu :

- **HTMLCanvasElement.getContext(contextType, contextAttributes)**

Method ini akan mengembalikan konteks menggambar pada *canvas*, atau *null* apabila *identifier* konteks tidak didukung. **Parameter:**

– **contextType**

tipe: **DOMString**

Berisi konteks yang menandakan konteks menggambar pada *canvas*. Parameter ini dapat berupa berbagai jenis nilai seperti berikut:

- * **"2d"**

Merepresentasikan konteks dua dimensi yang akan menciptakan objek *CanvasRenderingContext2D*.

- * **"webgl"**

Merepresentasikan konteks tiga dimensi yang akan menciptakan objek *WebGLRenderingContext*.

- * **"webgl2"**

Merepresentasikan konteks tiga dimensi yang akan menciptakan objek *WebGL2RenderingContext*.

Konteks ini hanya akan tersedia pada *browser* yang dapat mengimplementasi *WebGL* versi 2.

- * **bitmaprenderer**

Akan menciptakan objek *ImageBitmapRenderingContext* yang akan mengganti konten dari *canvas* dengan objek tersebut.

– **contextAttributes**

Berisi atribut dari parameter *contextType*. Contoh atribut dari beberapa tipe konteks yaitu sebagai berikut:

- * Atribut konteks *2d*:

· **alpha**

tipe: **boolean**

Menandakan apakah *canvas* berisi *alpha channel* atau tidak. Apabila bernilai *false*, maka *browser* akan menetapkan *backdrop* untuk selalu bernilai *opaque*, sehingga akan mempercepat proses menggambar.

- * Atribut konteks *WebGL*:

· **depth**

tipe: **boolean**

Menandakan apakah *buffer* untuk menggambar memiliki ukuran setidaknya 16 *bits* atau tidak.

· **antialias**

tipe: **boolean**

Menandakan apakah dapat melakukan proses *anti-aliasing* atau tidak.

Kembalian: Objek sesuai parameter *contextType*.

CanvasRenderingContext2D

Interface ini digunakan untuk menggambar persegi panjang, teks, gambar, dan objek-objek lain kedalam elemen *canvas*. *CanvasRenderingContext2D* menyediakan konteks *2D rendering* untuk suatu elemen *<canvas>*. Untuk mendapatkan objek dari *interface* ini, harus memanggil *getContext()* didalam elemen *<canvas>*, dengan memberi *"2d"* sebagai argumen. Berikut contoh penggunaannya :

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');
```

Properti yang dimiliki oleh *interfaces* ini terbagi menjadi beberapa bagian seperti berikut:

- i. **Fill dan Stroke styles**

- **CanvasRenderingContext2D.fillStyle**

Menentukan warna atau gaya yang akan digunakan pada bentuk tertentu. Nilai *default* dari properti ini yaitu #000 ('black'). Properti ini dapat berisi:

- **color**

tipe: **DOMString**

- **gradient**

tipe: **CanvasGradient**

- **pattern**

tipe: **CanvasPattern**

Contoh implementasi:

```
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');

// mengisi bentuk persegi dengan
// warna biru hanya pada sisinya
ctx.strokeStyle = 'blue';

// menggambar persegi tanpa
// ada warna didalam bentuknya
ctx.strokeRect(10, 10, 100, 100);
```

- **CanvasRenderingContext2D.strokeStyle**

Menentukan warna atau gaya yang akan digunakan pada garis sisi pada bentuk tertentu. Nilai *default* pada properti ini yaitu #000 ('black'). properti ini dapat berupa:

- **color**

tipe: **DOMString**

- **gradient**

tipe: **CanvasGradient**

- **pattern**

tipe: **CanvasPattern**

Contoh implementasi:

```
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');

// mengisi bentuk persegi dengan
// warna biru hanya pada sisinya
ctx.strokeStyle = 'blue';

// menggambar persegi tanpa
// ada warna didalam bentuknya
ctx.strokeRect(10, 10, 100, 100);
```

ii. *Line styles*

- **CanvasRenderingContext2D.lineWidth**

Menentukan ketebalan dari garis. Nilai *default* dari properti ini yaitu 1.0. Isi dari properti ini dapat berupa:

- **value**

Nilai yang menentukan ketebalan garis.

Contoh implementasi:

```
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');

ctx.lineWidth = 15;
```

- **CanvasRenderingContext2D.lineCap**

Menentukan jenis dari ujung suatu garis. Nilai dari properti ini dapat berupa:

- **butt**

Ujung dari garis memiliki bentuk rata.

- **round**

Ujung dari garis memiliki bentuk bulat.

- **square**

Ujung dari garis memiliki bentuk rata, ditambah kotak dengan ukuran lebar yang sama dan satu per delapan dari ketebalan garis tersebut.

Contoh implementasi:

```
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');

ctx.lineCap = 'round';
```

- **CanvasRenderingContext2D.lineJoin**

Menentukan bagaimana bentuk sudut dari kedua garis yang saling terhubung. Nilai dari atribut ini dapat berupa:

- **round**

Sudut memiliki bentuk bulat (melengkung).

- **bevel**

Sudut memiliki bentuk rata.

- **miter**

Sudut memiliki bentuk lancip.

Contoh implementasi:

```
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');

ctx.lineJoin = 'round';
```

iii. Text styles

- **CanvasRenderingContext2D.font**

Menentukan jenis teks yang akan digunakan. Nilai *default* dari properti ini yaitu 10px *sans-serif*.

Method yang dimiliki oleh *interfaces* ini terbagi menjadi beberapa bagian seperti berikut:

i. Menggambar *rectangles*

- **CanvasRenderingContext2D.clearRect(x, y, width, height)**

Method ini akan menghapus gambar sebelumnya dengan membentuk suatu persegi. *Method* ini akan menggambar koordinat titik awal (*x*, *y*) dengan lebar dan tinggi yang sudah ditentukan oleh *width* dan *height*.

Parameter:– **x**

Koordinat x yang menandakan titik awal persegi.

– **y**

Koordinat y yang menandakan titik awal persegi.

– **width**

Lebar persegi.

– **height**

tinggi persegi.

Contoh implementasi:

```
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');

ctx.fillRect(25, 25, 100, 100);

// Akan menghapus bagian dalam persegi yang
// sudah digambar sebelumnya
ctx.clearRect(45, 45, 60, 60);
```

• **CanvasRenderingContext2D.fillRect(x, y, width, height)**

Method ini akan menggambar persegi dengan warna tertentu didalam bentuknya. *Method* ini akan menggambar koordinat titik awal (x, y) dengan lebar dan tinggi yang sudah ditentukan oleh *width* dan *height*.

Parameter:– **x**

Koordinat x yang menandakan titik awal persegi.

– **y**

Koordinat y yang menandakan titik awal persegi.

– **width**

Lebar persegi.

– **height**

tinggi persegi.

Contoh implementasi:

```
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');

ctx.fillStyle = 'green';
ctx.fillRect(10, 10, 100, 100);
```

• **CanvasRenderingContext2D.strokeRect(x, y, width, height)**

Method ini akan menggambar persegi tanpa ada warna didalam bentuknya, namun akan memberi warna tertentu pada garis sisi persegi tersebut. *Method* ini akan menggambar koordinat titik awal (x, y) dengan lebar dan tinggi yang sudah ditentukan oleh *width* dan *height*.

Parameter:– **x**

Koordinat x yang menandakan titik awal persegi.

- **y**
Koordinat y yang menandakan titik awal persegi.

- **width**
Lebar persegi.
- **height**
tinggi persegi.

Contoh implementasi:

```
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');

ctx.strokeStyle = 'green';
ctx.strokeRect(10, 10, 100, 100);
```

ii. Paths

Paths merupakan kumpulan beberapa titik yang terhubung oleh garis yang dapat membentuk lengkungan, garis, atau bentuk tertentu. Untuk dapat membuat bentuk tertentu menggunakan *paths*, dapat dilakukan langkah berikut:

- A. Membuat *path*.
- B. Menggunakan beberapa fungsi *path* untuk menggambar suatu *path*.
- C. Menutup *path* yang sudah digambar.
- D. Setelah *path* ditutup, maka bentuk tersebut dapat diberi warna didalam bentuk tersebut maupun di garis sisinya.

Berikut merupakan beberapa *method* untuk membuat *path*:

- **CanvasRenderingContext2D.beginPath()**

Method ini digunakan untuk memulai suatu *path* baru dengan mengosongkan *list* dari *sub-paths* sebelumnya.

- **CanvasRenderingContext2D.moveTo(x, y)**

Method ini digunakan untuk memindahkan titik awal dari *sub-path* yang baru ke koordinat (x, y) .

Parameter:

- **x**
Koordinat x yang menandakan titik pada posisi sumbu x.

- **y**
Koordinat y yang menandakan titik pada posisi sumbu y.

- **CanvasRenderingContext2D.lineTo(x, y)**

Method ini digunakan untuk menghubungkan titik sebelumnya pada *sub-path* dengan koordinat (x, y) . **Parameter:**

- **x**
Koordinat x yang menandakan akhir dari garis.

- **y**
Koordinat y yang menandakan akhir dari garis.

- **CanvasRenderingContext2D.closePath()**

Method ini berfungsi untuk memindahkan posisi titik saat ini kembali ke posisi awal *sub-path*. Apabila bentuk yang dibuat oleh *path* sudah ditutup, atau hanya memiliki satu titik, maka *method* ini tidak akan berfungsi.

- **CanvasRenderingContext2D.arc(x, y, radius, startAngle, endAngle, anticlockwise)**

Method ini akan menambah lengkungan pada *path* dengan titik tengah berada pada posisi (x, y) , memiliki *radius* dan dimulai dari sudut *startAngle* dan berakhir di sudut *endAngle*, dengan cara menggambar sesuai arah *anticlockwise*.

Parameter:

– **x**

Koordinat x dari titik tengah lengkungan.

– **y**

Koordinat y dari titik tengah lengkungan.

– **radius**

Radius lengkungan.

– **startAngle**

Derasat awal dari lengkungan, diukur searah jarum jam dari posisi sumbu x positif dalam radian.

– **endAngle**

Derasat akhir dari lengkungan, diukur searah jarum jam dari posisi sumbu x positif dalam radian.

– **anticlockwise**

Cara menggambar lengkungan. Apabila bernilai *true*, maka akan digambar dengan cara berlawanan arah jarum jam.

Contoh implementasi:

```
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');

// langkah berikut akan membuat gambar segitiga
ctx.beginPath();
ctx.moveTo(20, 20);
ctx.lineTo(200, 20);
ctx.lineTo(120, 120);
ctx.closePath();
ctx.stroke();
```

2. Menganalisis aplikasi sejenis.

status : Ada sejak rencana kerja skripsi.

hasil :

Salah satu aplikasi sejenis permainan berbasis web dengan memanfaatkan *smartphone* sebagai pengendali yaitu AirConsole [?]. Aplikasi tersebut memanfaatkan teknologi *browser*, *smartphone*, *PC*, dan juga jaringan internet untuk dapat menggunakananya. Aplikasi ini dikembangkan oleh N-Dream AG [?].

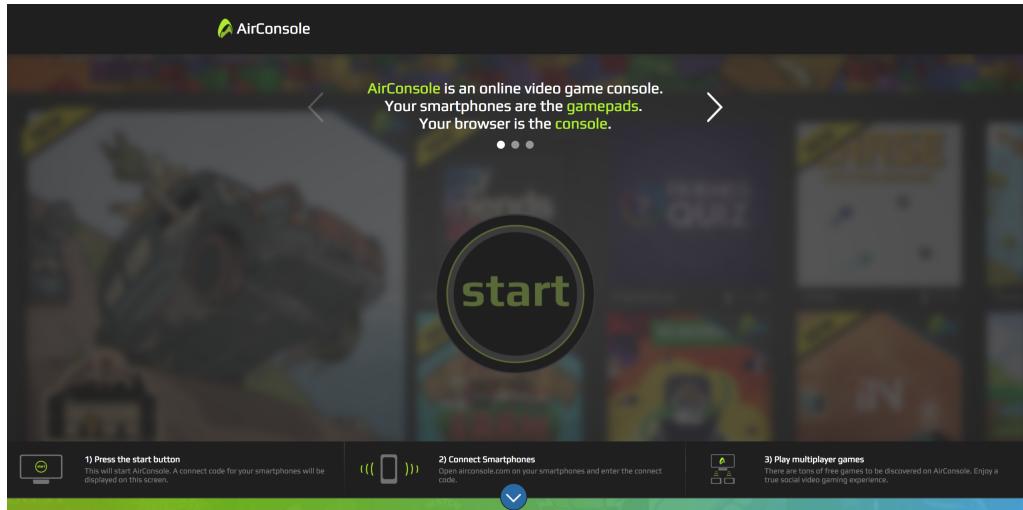
Analisis AirConsole

AirConsole merupakan permainan berbasis web dimana *browser* pada *mobile device* dapat melakukan koneksi ke *browser* pada *PC*. Pada aplikasi ini, terdapat berbagai macam permainan yang dapat dipilih oleh pemain. Untuk dapat memainkan aplikasi tersebut, pemain harus membuka alamat web *airconsole.com* pada *PC browser* dan juga pada *smartphone browser*.

Analisis dilakukan dengan cara berikut:

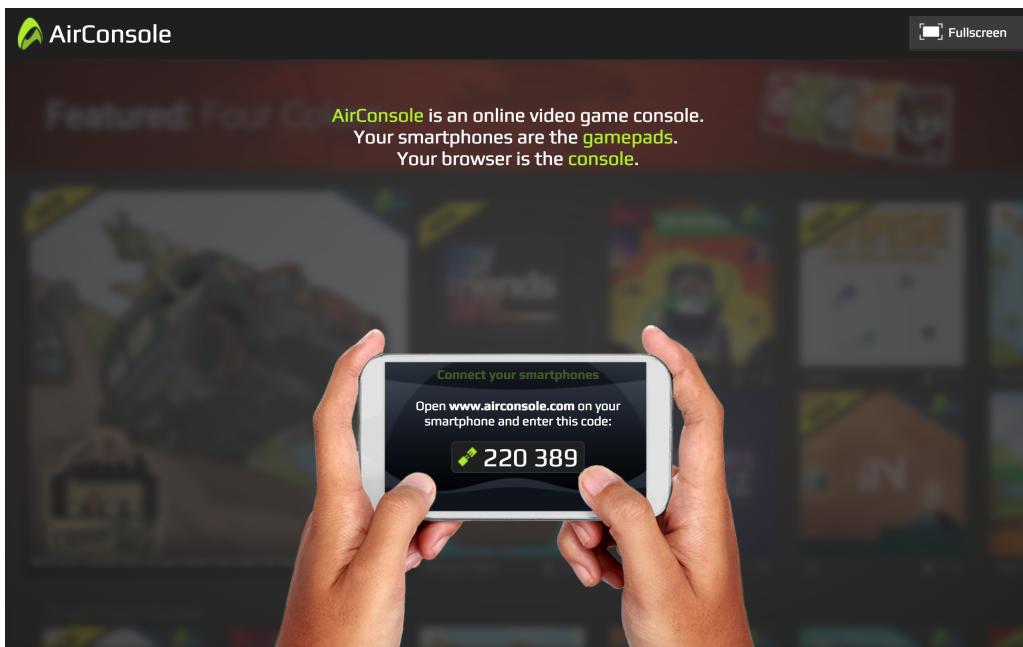
- (a) Memainkan permainan dari awal hingga akhir.
- (b) Keluar dari *browser* pada *PC* pada saat permainan berlangsung.
- (c) Keluar dari *browser* pada *smartphone* pada saat permainan berlangsung.

Pada halaman awal web di *PC*, pemain diminta untuk menekan tombol *start* yang ada pada gambar berikut:



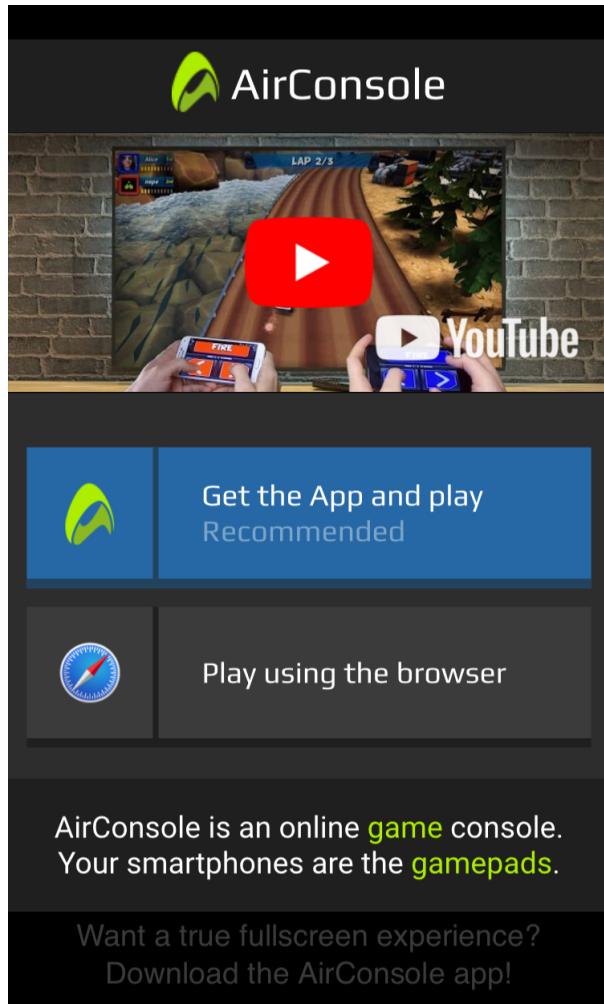
Gambar 1: Halaman awal web pada *PC browser*.

Setelah tombol *start* ditekan, maka akan muncul halaman berikutnya yang menunjukkan kode yang harus dimasukan oleh pemain pada *mobile browser*.



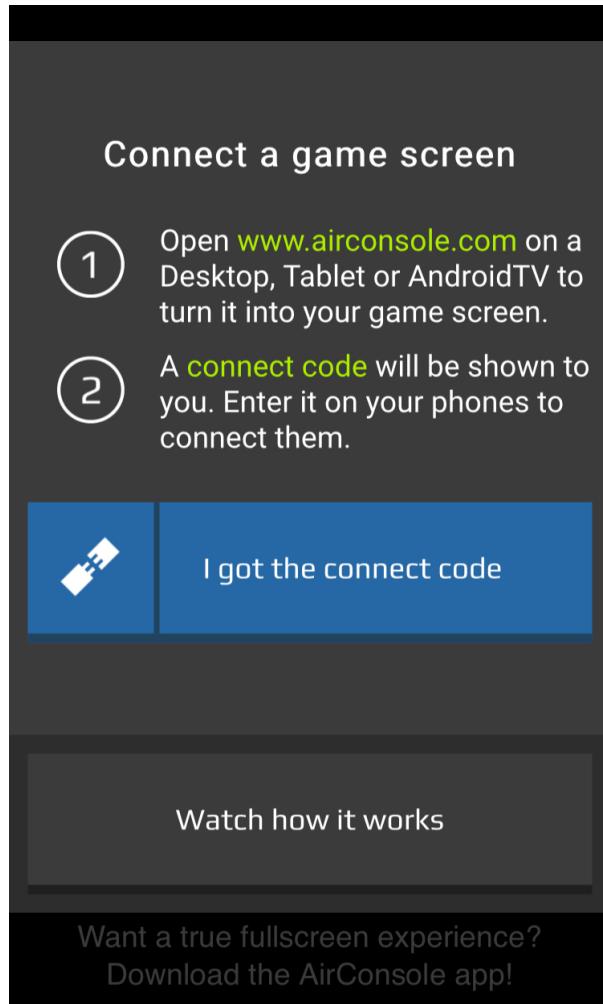
Gambar 2: Kode yang harus dimasukan oleh pemain pada *mobile browser*.

Pemain harus mengakses alamat web yang sama pada *mobile browser*. Pada halaman awal, pemain akan diminta untuk memilih apakah akan bermain dengan menggunakan aplikasi, atau bermain dengan menggunakan *browser*.



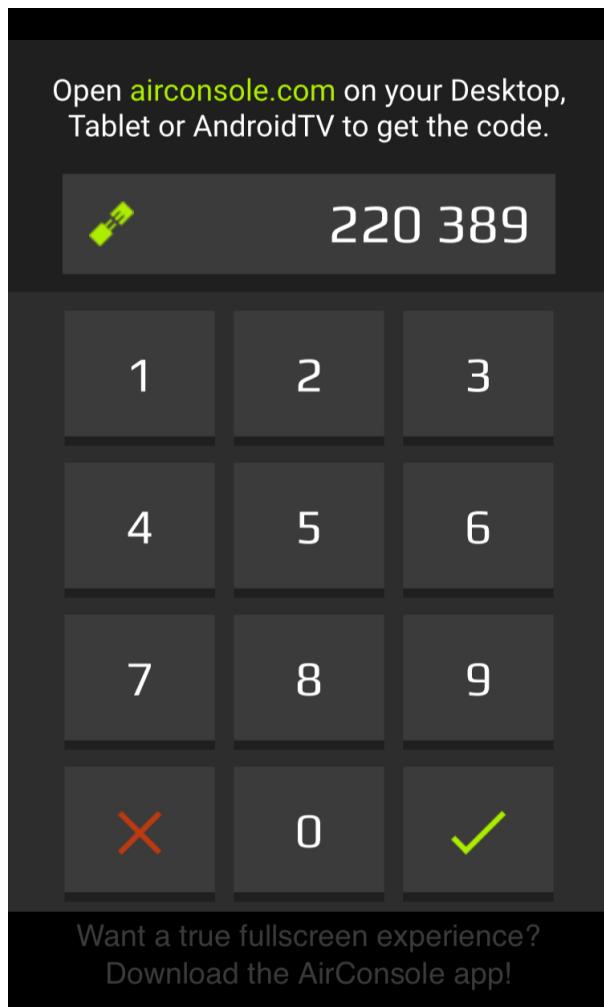
Gambar 3: Halaman awal pada *mobile browser*.

Dalam analisis ini, penulis memilih untuk bermain menggunakan *browser*. Setelah itu, pemain diminta untuk menekan tombol '*i got the connect code*' untuk memasukan kode yang sudah didapatkan pada *PC browser*.



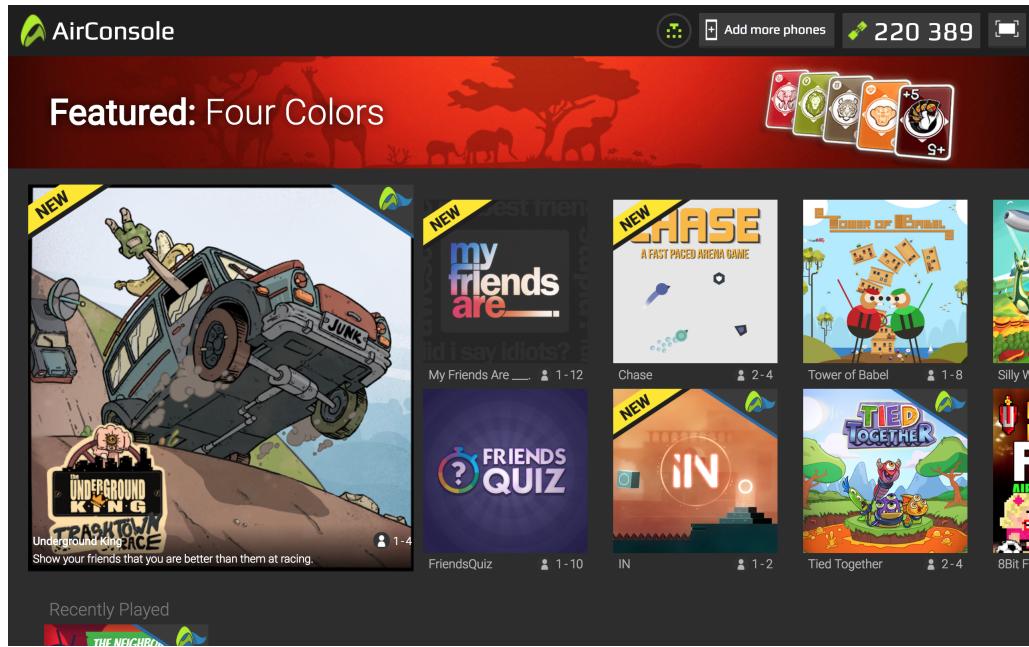
Gambar 4: Pemain diminta untuk memasukan kode yang sudah didapatkan pada *PC browser*.

Setelah menekan tombol tersebut, pemain dapat mulai memasukan kode yang sudah didapatkan. Kode ini bertujuan untuk proses otentikasi, sehingga para pemain yang dapat bermain dalam satu sesi yang sama, hanya para pemain yang mengetahui kode tersebut.

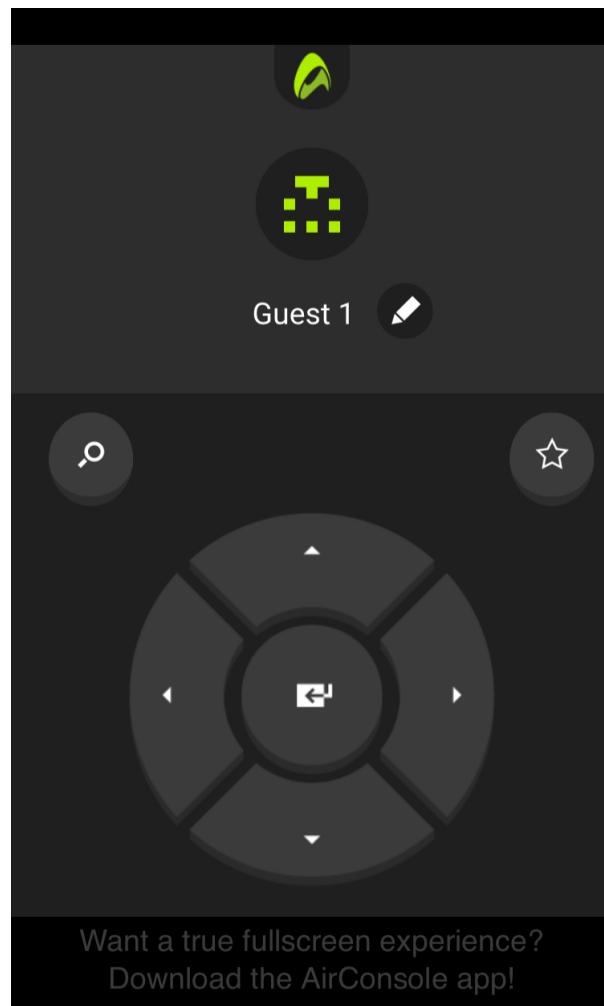


Gambar 5: Pemain diminta untuk memasukan kode yang sudah didapatkan pada *PC browser*.

Setelah pemain memasukan kode, maka halaman web di *PC* dan *smartphone* akan berubah. Pada *PC*, halaman akan menunjukan berbagai jenis permainan yang dapat dipilih. Pada *smartphone*, halaman akan berubah menjadi pengendali permainan, dimana pemain dapat menggerakan halaman yang ada di *PC* dengan menggunakan *smartphone*.



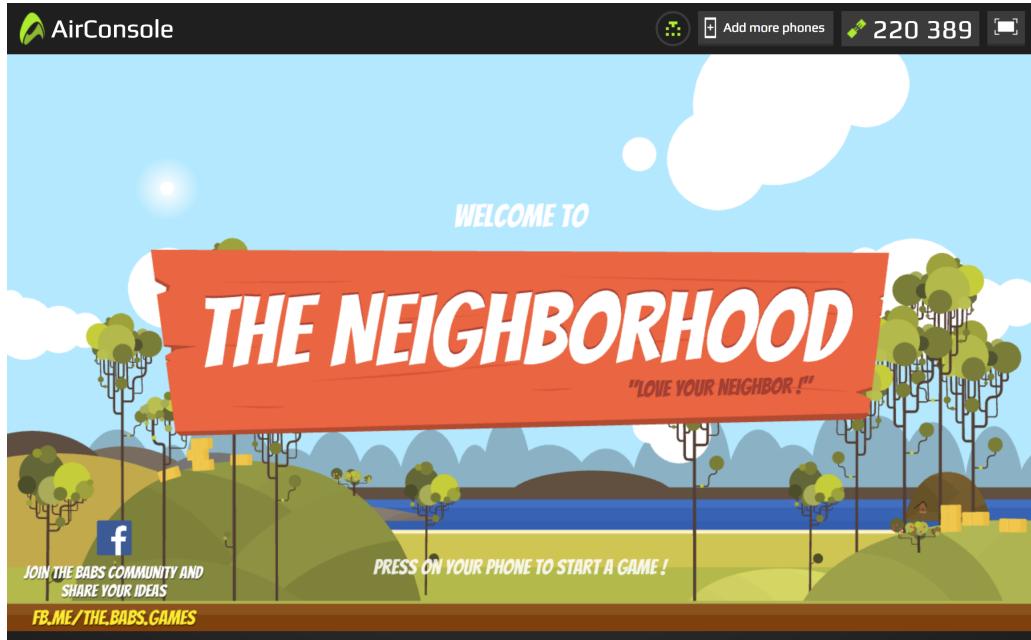
Gambar 6: Halaman pada *PC* yang menunjukkan berbagai permainan yang dapat dipilih.



Gambar 7: Halaman pada *smartphone* yang berfungsi sebagai pengendali.

Dalam analisis ini, penulis memilih untuk memainkan permainan yang bernama *The Neighborhood*.

Permainan ini sejenis permainan Angry Birds [?]. Permainan ini bercerita tentang dua kelompok yang bertetangga, dimana kelompok tersebut bermusuhan dan berusaha untuk saling menghancurkan satu sama lain. Tujuan dari permainan ini yaitu lebih dulu menghancurkan anggota kelompok tetangga. Setelah memilih permainan tersebut, halaman pada *PC* dan *smartphone* akan berubah. Pada *smartphone*, pemain akan diminta untuk merubah mode tampilan *smartphone* menjadi *landscape*.

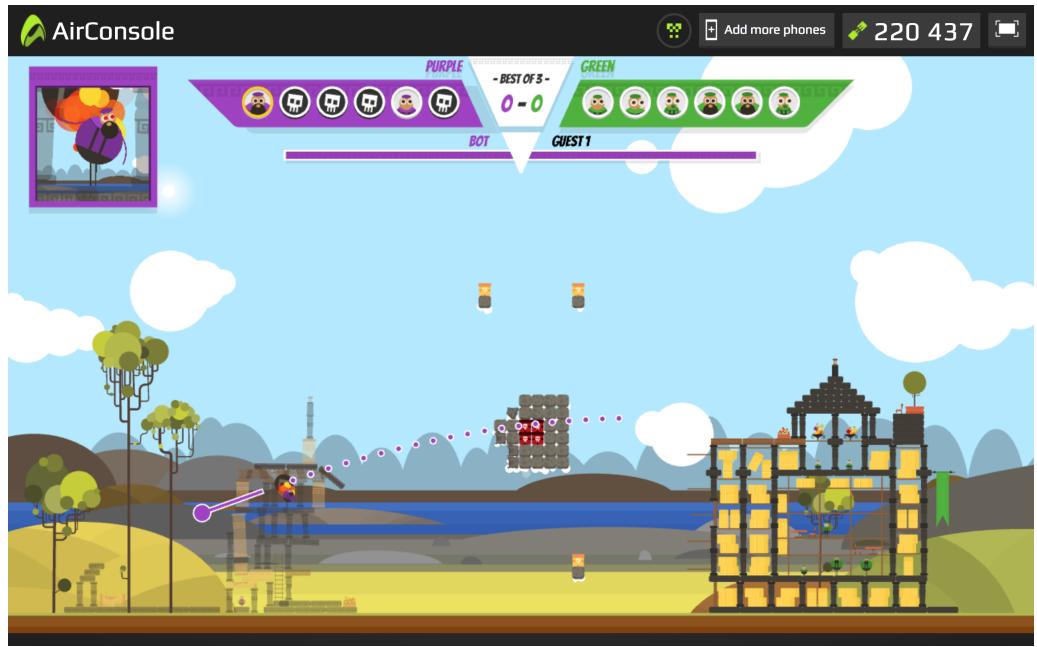


Gambar 8: Halaman awal permainan The Neighborhood pada *PC*.

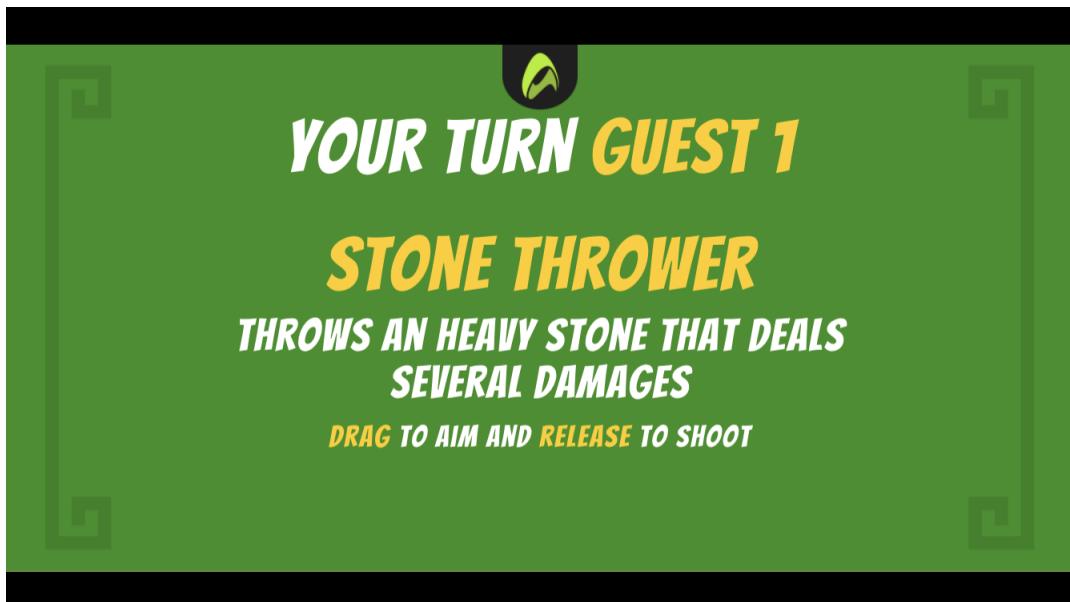


Gambar 9: Halaman awal permainan The Neighborhood pada *smartphone*.

Cara bermain dari permainan tersebut yaitu dengan menggunakan *smartphone*, dimana pemain harus menekan layar *smartphone*, kemudian menariknya sesuai dengan arah yang berlawanan dengan lawan, lalu melepas jari dari layar *smartphone* dengan tujuan untuk melempar suatu benda dari ketapel. Semakin jauh pemain menarik, maka lontaran benda tersebut akan semakin kencang mengenai lawan.

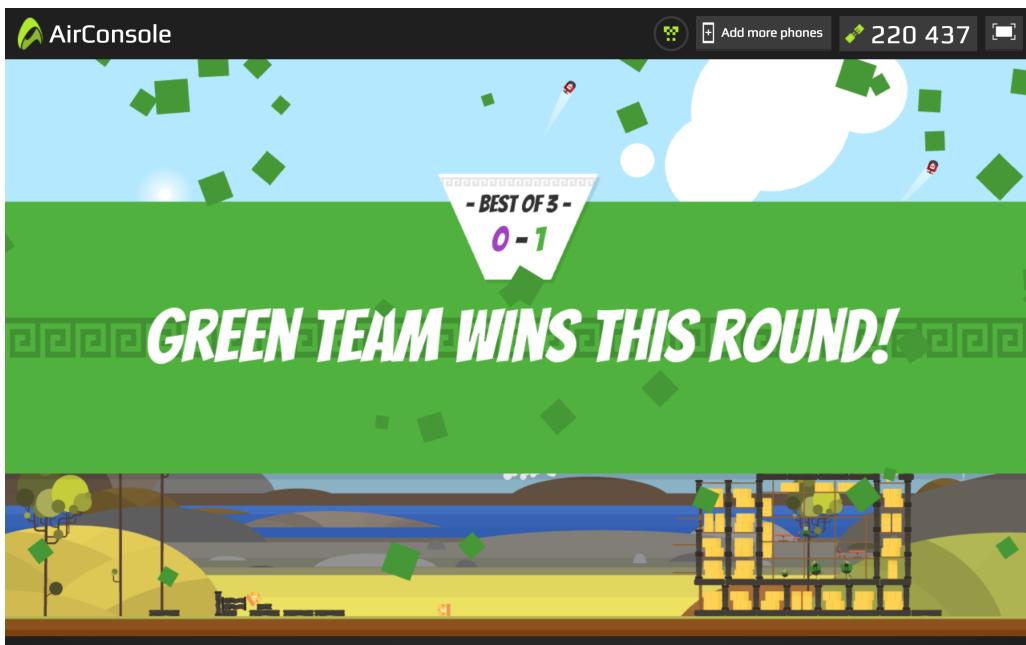


Gambar 10: Halaman pada *PC* dimana permainan sedang berlangsung.

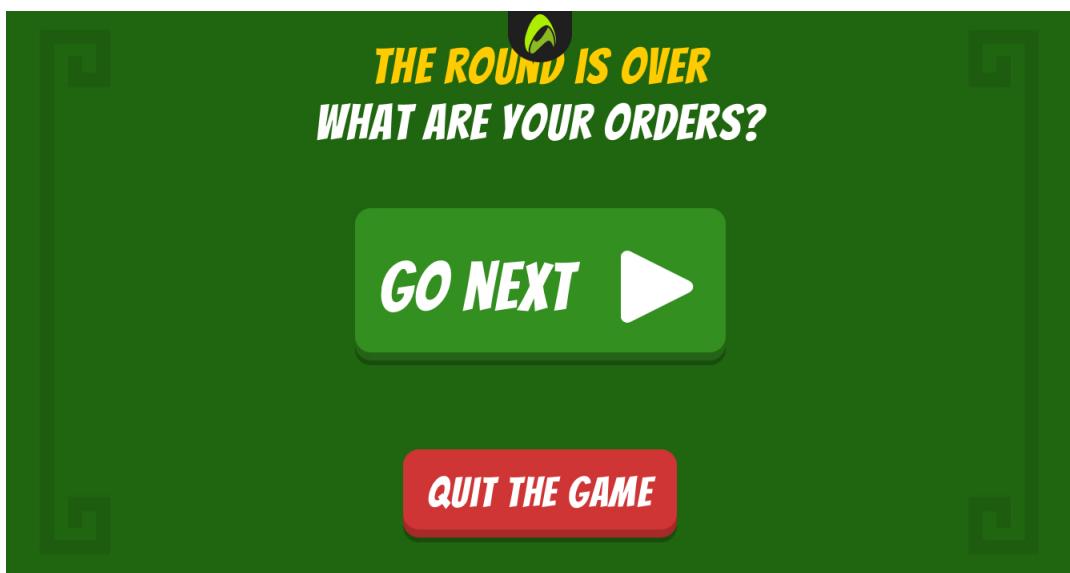


Gambar 11: Halaman pada *smartphone* dimana permainan sedang berlangsung.

Apabila memenangkan permainan tersebut, maka pemain dapat memilih untuk keluar dari permainan atau melanjutkan permainannya kembali.

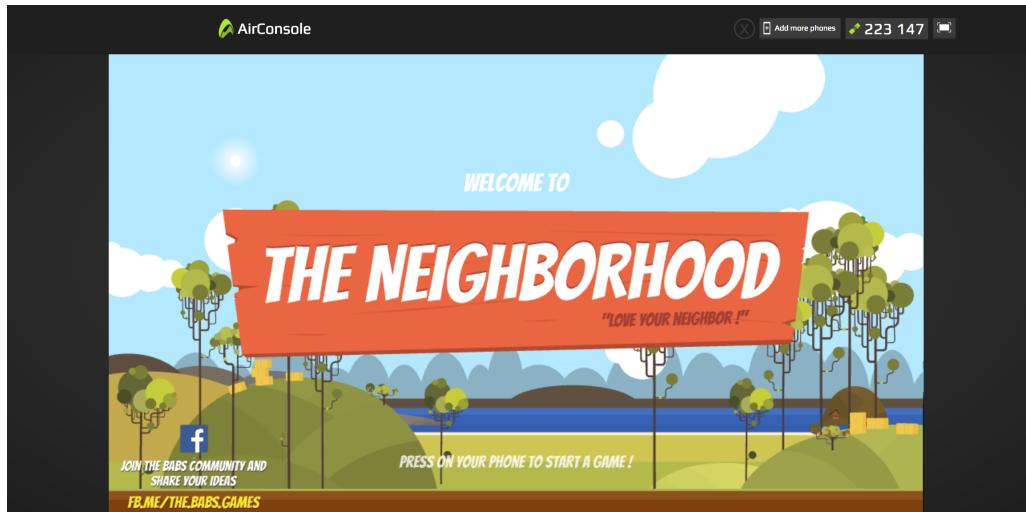


Gambar 12: Halaman pada *PC* apabila permainan sudah dimenangkan.



Gambar 13: Halaman pada *smartphone* apabila permainan sudah dimenangkan.

Dari ketiga percobaan yang sudah dilakukan, ada beberapa hal yang dapat diperbaiki dari permainan berbasis web tersebut. Percobaan pertama menunjukkan hasil yang bagus, dimana koneksi antara *smartphone* dan *PC* tidak putus saat permainan berlangsung, dan juga tidak ada keterlambatan antara gerakan pada *smartphone* dan *PC*. Pada percobaan kedua, apabila *browser* pada *PC* ditutup pada saat permainan berlangsung, maka koneksi akan terputus. Tetapi, tampilan pada *smartphone* tidak menunjukkan bahwa adanya koneksi yang terputus, sehingga pemain tidak mengetahui apakah permainan masih dapat berlangsung atau tidak. Tampilan hanya akan langsung kembali pada halaman awal permainan. Begitupun dengan percobaan ketiga, apabila *browser* pada *smartphone* ditutup pada saat permainan sedang berlangsung, maka koneksi akan terputus. Tampilan pada *PC* hanya menunjukkan tanda kecil bahwa telah terjadi pemutusan koneksi pada *smartphone*, yaitu tanda x pada bagian atas tampilan yang ditunjukkan seperti gambar berikut:



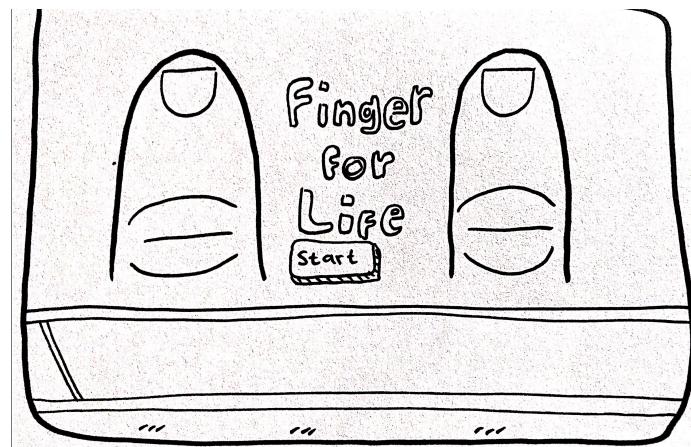
Gambar 14: Halaman pada *PC* yang menunjukan pemutusan koneksi.

Perbaikan yang dapat dilakukan adalah dengan memberi *feedback* yang lebih jelas pada pemain, apabila ada kesalahan pada aplikasi yang terjadi seperti pemutusan koneksi. Dengan begitu, pemain akan lebih mengetahui bahwa koneksi dapat saja terputus dan tidak dapat melanjutkan permainannya.

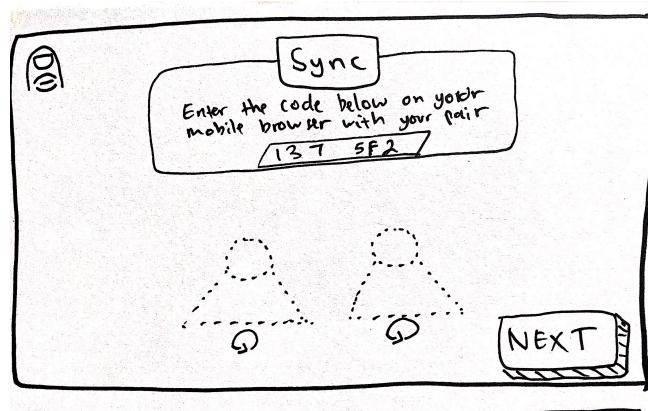
3. Merancang antarmuka permainan pada *PC* dan *smartphone*.
status : Ada sejak rencana kerja skripsi.
hasil :

Antarmuka yang dirancang terbagi menjadi dua bagian, yaitu antarmuka pada *browser* yang ada di *PC* dan *smartphone*.

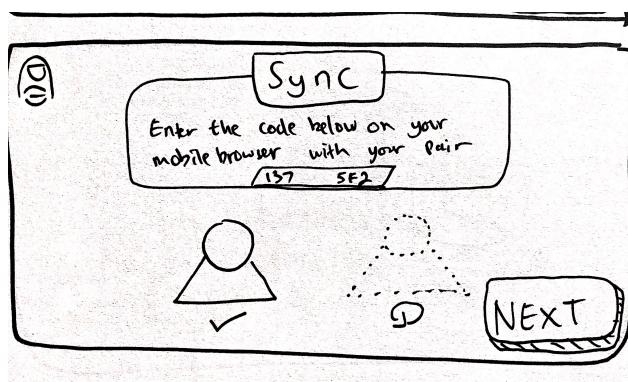
PC



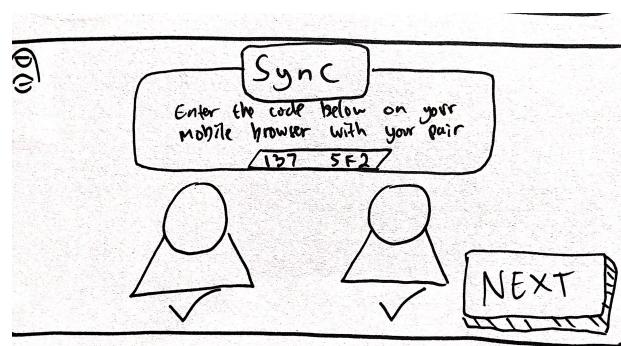
Gambar 15: Tampilan awal web



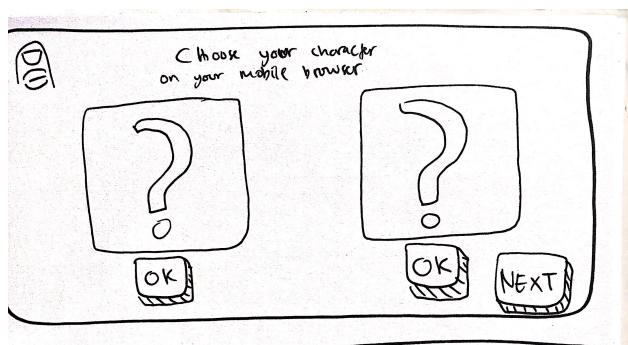
Gambar 16: Para pemain yang akan melakukan proses *sync*



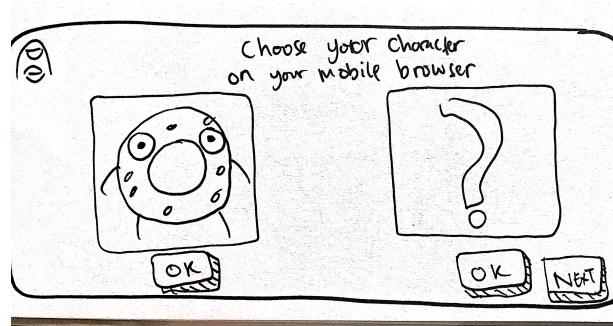
Gambar 17: Pemain pertama yang sudah melakukan proses *sync*



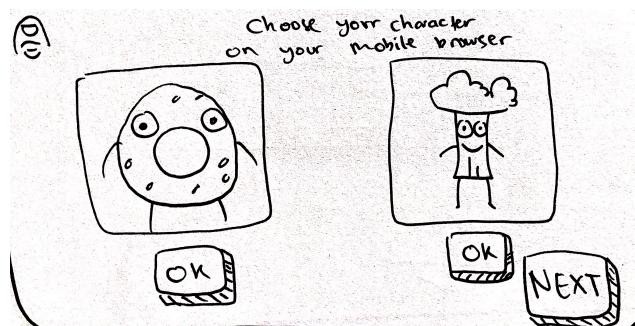
Gambar 18: Pemain kedua yang sudah melakukan proses *sync*



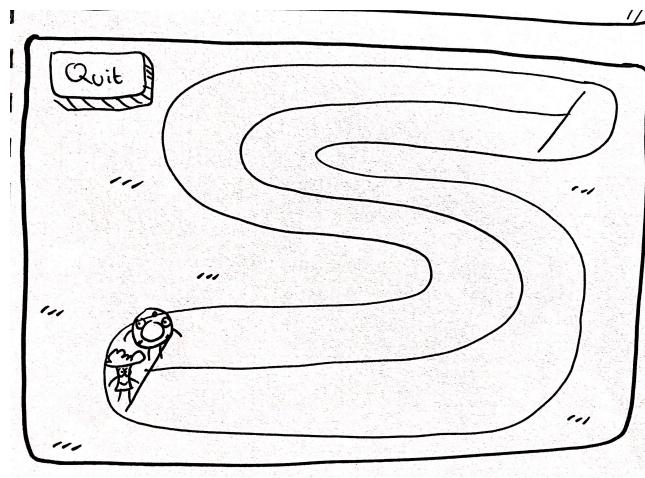
Gambar 19: Para pemain yang akan memilih karakter



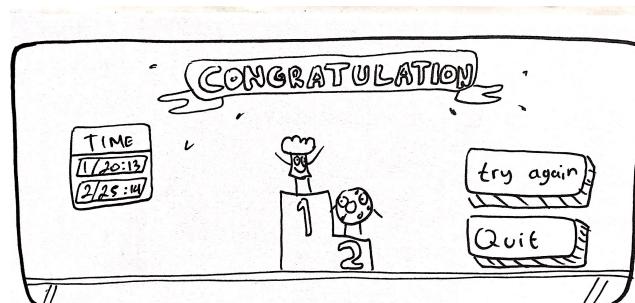
Gambar 20: Pemain pertama yang sudah memilih karakter.



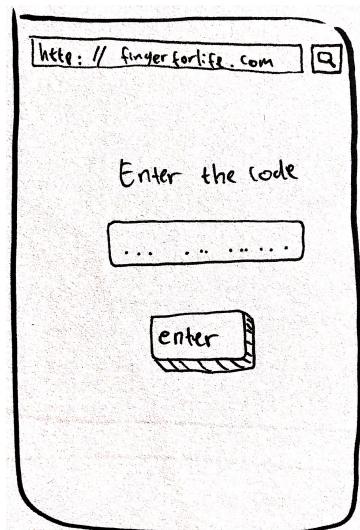
Gambar 21: Pemain kedua yang sudah memilih karakter.



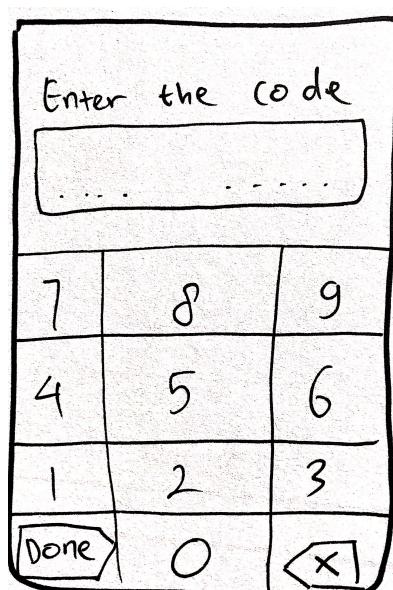
Gambar 22: Kedua pemain yang akan siap memainkan permainan.



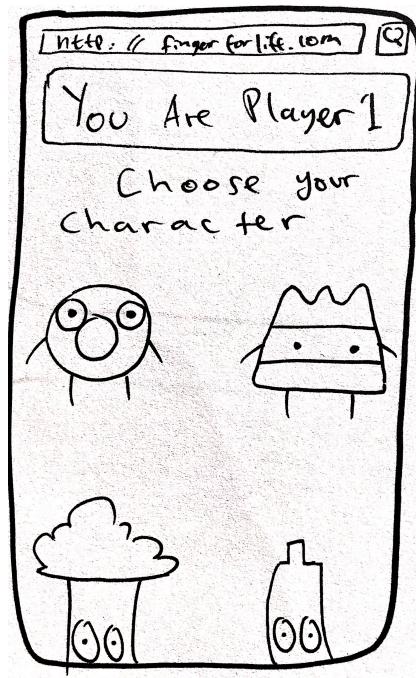
Gambar 23: Permainan sudah selesai

SMARTPHONE

Gambar 24: Pemain mengakses halaman web pada *mobile browser*.



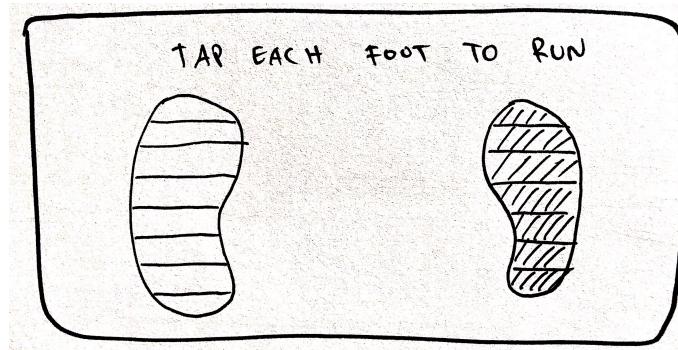
Gambar 25: Pemain memasukan kode yang didapatkan pada halaman web di *PC*.



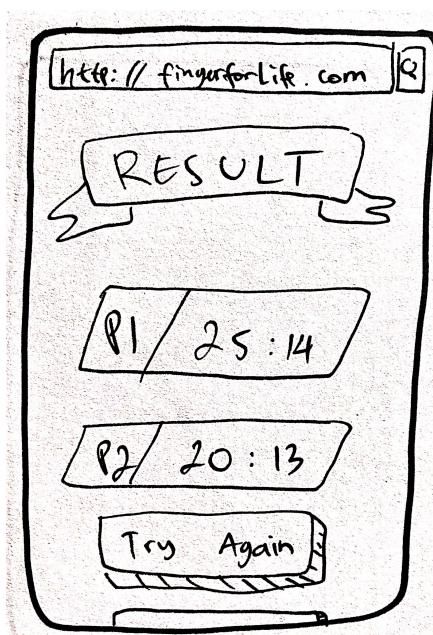
Gambar 26: Pemain memilih karakter yang akan dimainkan.



Gambar 27: Pemain menentukan karakter.



Gambar 28: Pemain mulai memainkan permainan.



Gambar 29: Permainan berakhir.

4. Menyusun cara bermain aplikasi permainan yang dibangun.

status : Ada sejak rencana kerja skripsi.

hasil :

Pada skripsi ini, akan dibuat sebuah aplikasi permainan yang memanfaatkan protokol *WebSockets*, dimana dalam penggunaan protokol tersebut akan dibantu dengan teknologi *Socket.io*. Selain itu, aplikasi yang dibuat akan memanfaatkan *personal computer (PC)* dan *smartphone* untuk pengembangan aplikasinya. Kedua teknologi tersebut merupakan teknologi yang sudah dimiliki oleh banyak orang. Oleh karena itu, aplikasi permainan yang akan dibangun akan memanfaatkan *PC* dan *smartphone*.

Nama permainan yang akan dibangun adalah *Finger For Life*. Permainan tersebut merupakan adu balap lari yang dapat dimainkan oleh dua orang pemain, dimana para pemain akan memiliki karakter untuk dimainkan pada trek lari yang berbentuk huruf S di layar *PC*. Agar dapat memainkan permainan tersebut, para pemain harus memiliki *smartphone* dan *PC* beserta koneksi internet yang stabil. Apabila hal-hal tersebut terpenuhi, pemain dapat membuka *web browser* pada *PC* untuk mengakses alamat web yang akan menuju ke permainan *Finger For Life*. Para pemain akan diminta untuk melakukan dua hal agar dapat memainkan permainan tersebut bersama seorang rekan yang akan menjadi lawan mainnya, yaitu :

- Membuka *web browser* pada *PC* untuk mengakses alamat web permainan *Finger For Life*.

- Mengakses alamat web permainan dan memasukan kode tertentu pada *web browser* di *smartphone* untuk sesi permainan saat ini.

Kedua hal tersebut bertujuan untuk melakukan koneksi antara *smartphone* dan *PC*, dimana *smartphone* akan berfungsi sebagai *controller* dalam permainan. Apabila kedua hal diatas telah dilakukan, maka kedua pemain akan dapat mulai memainkan permainannya. Permainan akan diawali dengan pemilihan karakter oleh kedua pemain, dimana karakter tersebut akan berfungsi sebagai representasi masing-masing pemain dalam permainan *Finger For Life*. Setelah pemilihan karakter selesai, maka para pemain akan dibawa ke halaman selanjutnya yang berupa halaman *game on*. Pada halaman ini, para pemain diminta untuk memegang *smartphone* masing-masing untuk mencoba memainkan permainannya dengan cara menekan tombol-tombol yang muncul pada *smartphone*. Hal tersebut bertujuan agar para pemain terbiasa terlebih dahulu dengan cara bermainnya. Setelah hal itu dilakukan, maka para pemain dapat memulai memainkan permainannya.

Para pemain akan mengkoneksikan *smartphone* miliknya pada suatu *PC*, dimana *smartphone* tersebut akan berfungsi sebagai *controller* untuk memainkan permainannya. Oleh karena itu, protokol *Web-Sockets* akan digunakan sebagai koneksi antara *smartphone* dan *PC* dalam aplikasi permainan yang akan dibangun. Aplikasi permainan akan menggunakan teknologi berbasis web, sehingga untuk memainkannya, *client* bisa mengakses melalui *web browser* tanpa harus berada di satu jaringan lokal yang sama.

5. Mengimplementasi program aplikasi permainan berbasis web.

status : Ada sejak rencana kerja skripsi.

hasil : -

6. Menganalisis *latency* yang dihasilkan pada aplikasi.

status : Ada sejak rencana kerja skripsi.

hasil : -

7. Melakukan eksperimen dan pengujian yang melibatkan responden.

status : Ada sejak rencana kerja skripsi.

hasil : -

8. Menulis dokumen skripsi

status : Ada sejak rencana kerja skripsi.

hasil : Dokumen sudah ditulis dari bab1 hingga bab2.

Pustaka

- [1] Dahl, R. (2009) Node.js. <https://nodejs.org/en/>. [Online; diakses 7-Oktober-2017].
- [2] Holowaychuk, T. (2010) Express.js. <https://expressjs.com/>. [Online; diakses 7-Oktober-2017].
- [3] Mozilla (2011) WebSockets. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. [Online; diakses 7-Oktober-2017].
- [4] Rauch, G. (2011) Socket.io. <https://socket.io/>. [Online; diakses 7-Oktober-2017].
- [5] Rauch, G. (2011) Socket.io Server API. <https://socket.io/docs/server-api/>. [Online; diakses 7-Oktober-2017].
- [6] Mozilla (2004) Canvas API. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API. [Online; diakses 7-Oktober-2017].

3 Pencapaian Rencana Kerja

Persentase penyelesaian skripsi sampai dengan dokumen ini dibuat dapat dilihat pada tabel berikut :

1*	2*(%)	3*(%)	4*(%)	5*	6*(%)
1	10	10			10
2	5	5			1
3	10	5	5		5
4	15	5	10		5
5	20	5	15		1
6	10		10		
7	10		10		
8	20	10	10	6 menulis dokumen skripsi dari bab1 hingga bab3 di S1	
Total	100	40	60		28

Keterangan (*)

1 : Bagian penggerjaan Skripsi (nomor disesuaikan dengan detail penggerjaan di bagian 5)

2 : Persentase total

3 : Persentase yang akan diselesaikan di Skripsi 1

4 : Persentase yang akan diselesaikan di Skripsi 2

5 : Penjelasan singkat apa yang dilakukan di S1 (Skripsi 1) atau S2 (skripsi 2)

6 : Persentase yang sidah diselesaikan sampai saat ini

Bandung, 22/11/2017

Priambodo Pangestu

Menyetujui,

Nama: Pascal Alfadian Nugroho

Pembimbing Tunggal