SEBA Master: Web Application Engineering

Advanced Topics in Web Application Engineering

Prof. Dr. Florian Matthes, Munich

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

Outline

7. Advanced topics in web application engineering

- Transcompilers for the web
- Real-time web applications
- Hybrid web applications

© sebis

Transcompilable tools for the web

- **Transcompilation** or **Transpilation** refers to the process of translating source code from one language to another one on the **same level of abstraction**
  - cf. **Compilation**: translating source code from a higher-level language to a lower-level one

- Purpose of transcompilation and transcompilable tools:
  + Translating **legacy code** to current version of language
  + „**Syntactic sugar**“ which is transcompiled to regular code
  + Support for additional **compile-time features**
    - Examples: Static typing, OO features, global variables, etc.
  + Improvements of **source code organization** and **maintenance**
    - Example: Source code can be organized in maintainable components, which are transcompiled to a monolithic and executable structure)
  + Better **IDE support**

- Drawbacks
  - Additional programming language requires additional knowledge and training
  - More complex build process, deployment, and debugging
  - Impeded integration of libraries and frameworks written in destination languages

© sebis

Why transcompilable tools for the web?

- Releasing new versions of standardized languages (e.g., HTML, CSS, JavaScript) takes usually very long, and is a very **cumbersome process** involving **many stakeholders**
- It takes years until the design of a such a language is adapted to the ever-changing requirements
  - Example: In recent years, the front-end of web application became bigger and bigger. However, JavaScript suffers from shortcomings for the development of large-scale web applications due to the lack of concepts like modules, classes, static typing, etc.

- Transcompilation tools can provide useful language features **without** having to **adapt the actual destination language**

- **Different transcompilation tools** for the same destination language can provide **different features** for different needs

© sebis

Transcompilable languages for HTML

- Generation of valid HTML markup which can be interpreted by prevalent web browsers

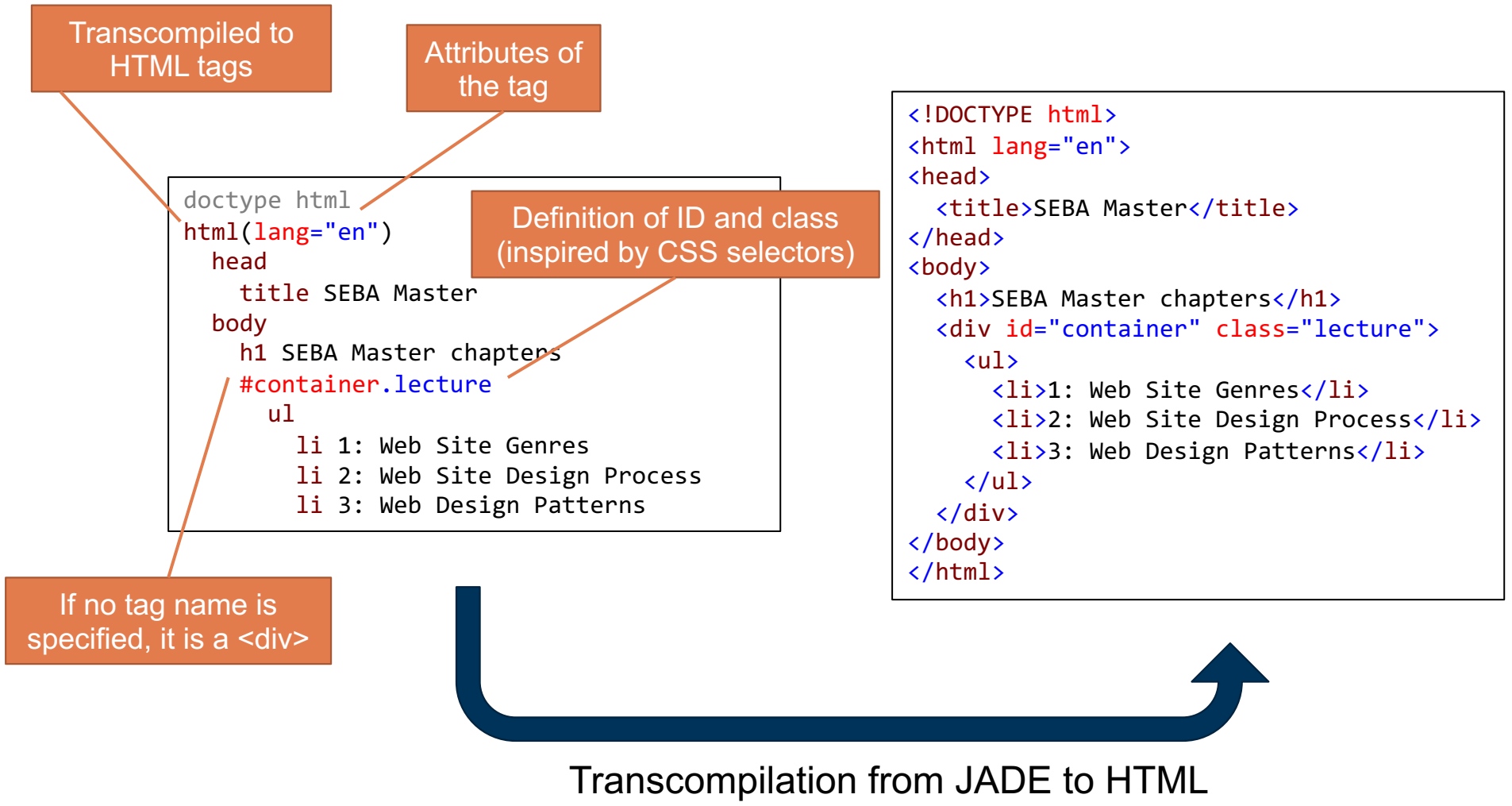
- Typical features of transcompilable languages for HTML:
  - + Concise syntax
  - + Generation of well-formed HTML markup (e.g., generation of missing closing tags)
  - + Support for common programming language concepts, e.g., iterations, conditionals, etc.
  - + Partition of HTML pages into reusable components

- Often used to define HTML **templates** which are, e.g., initiated by JavaScript

© sebis

Transcompilable languages for HTML

Example: JADE



http://jade-lang.com/

© sebis

Transcompilable languages for CSS

- Generation of valid CSS stylesheets which can be interpreted by prevalent web browsers

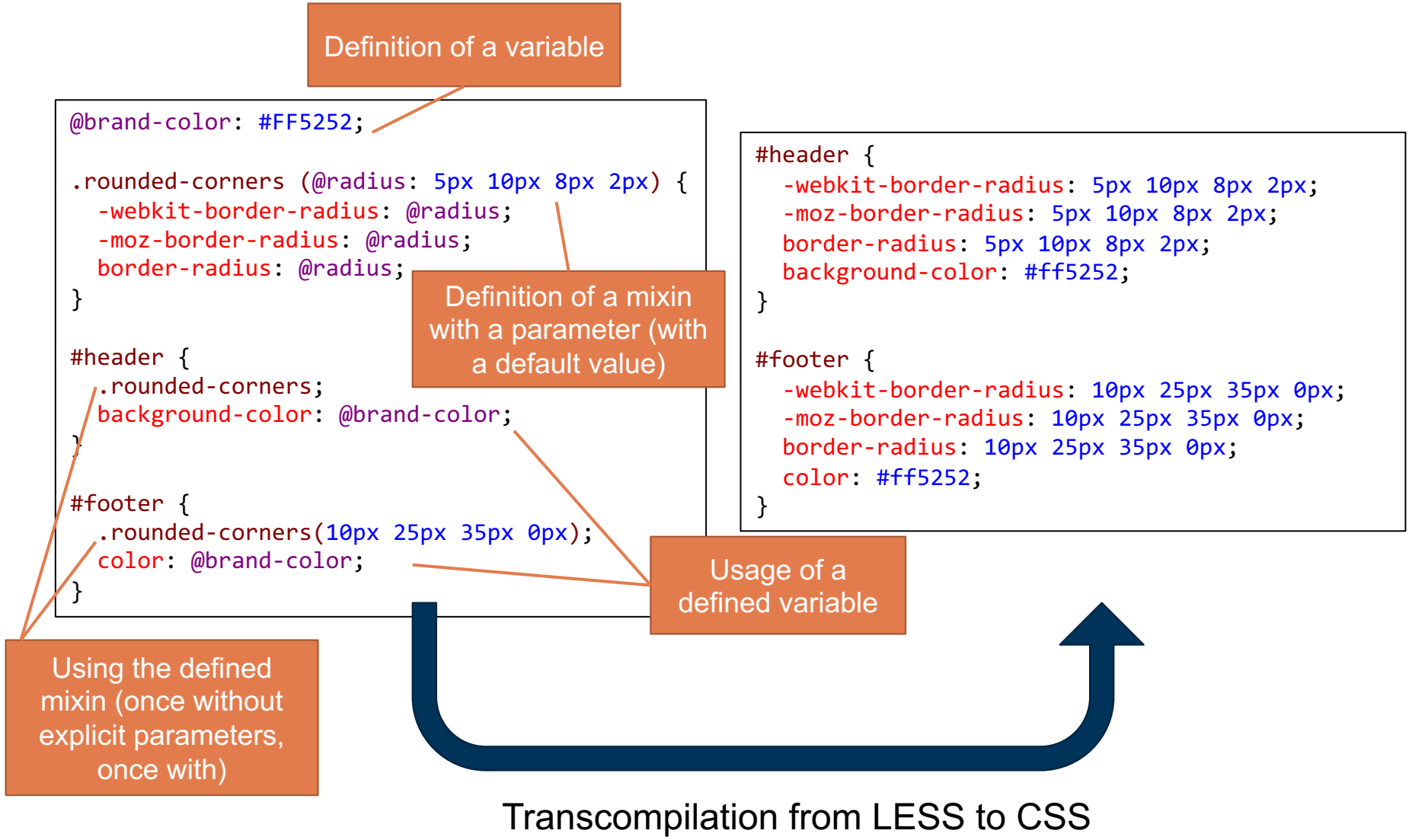
- Typical features of transcompilable languages for JavaScript:
  + Definition of variables (e.g., for colors)
  + Nested/Hierarchical structuring of stylesheets
  + Composition of style classes by mixins, i.e., properties of a style class can be embedded in another one

- Transcompilable languages for CSS are also called „**dynamic style-sheet languages**“
  - Some of them can even be transcompiled dynamically on the client
  - Most common examples: SASS and LESS

© sebis

Transcompilable languages for CSS

Example: LESS



http://lesscss.org/

© sebis

Transcompilable languages for JavaScript

- Generation of valid JavaScript code which can be interpreted by prevalent web browsers

- Typical features of transcompilable languages for JavaScript:
  + Support for classes and modules
  + Type annotations for static typing
  + Syntactic sugar for better readability (e.g., for lambda expressions)
  + Focus on specific language paradigms (e.g., functional programming)
  + Abstractions for asynchronous programming

- The development of some transcompilable language for JavaScript is driven by software vendors, e.g., TypeScript by Microsoft, and Dart by Google
  - Angular 2.0 natively supports both TypeScript and Dart

- Other related languages: CoffeeScript, LiveScript, and much more.

© sebis

Transcompilable languages for JavaScript

Example: TypeScript

```
class Person {
  private name: string;
  private age: number;
  private gender: Gender;

  constructor(name: string, age: number,
              gender: Gender) {
    this.name = name;
    this.age = age;
    this.gender = gender;
  }

  getYearOfBirth(): number {
    return new Date().getFullYear() - this.age;
  }
}

enum Gender {
    male, female
}
```

Definition of a class

(Optional) type annotations

Constructor and methods are transcompiled into JavaScript Prototypes

Support for prevalent language constructs (e.g., enumerations)

```
var Person = (function () {
  function Person(name, age, gender) {
    this.name = name;
    this.age = age;
    this.gender = gender;
  }
  Person.prototype.getYearOfBirth = function () {
    return new Date().getFullYear() - this.age;
  };
  return Person;
})();

var Gender;
(function (Gender) {
  Gender[Gender["male"] = 0] = "male";
  Gender[Gender["female"] = 1] = "female";
})(Gender || (Gender = {}));
```

Transcompilation from TypeScript to JavaScript (ECMAScript 5)

http://www.typescriptlang.org/

© sebis

Outline

7. Advanced topics in web application engineering

- Transcompilers for the web
- Real-time web applications
- Hybrid web applications

© sebis

What are real-time web applications

- A set of technologies and practices that enable users to receive information as soon as it is published by its authors, rather than requiring that they or their software check a source periodically for updates.
- The information types transmitted this way are often short messages, status updates, news alerts, or links to longer documents. The content is often "soft" in that it is based on the social web—people's opinions, attitudes, thoughts, and interests—as opposed to “hard” content such as entire documents with news or facts.

Use cases

- Instant messaging applications (e.g. Facebook Messenger, WhatsApp)
- Social activity streams (e.g. The Facebook feed)
- Multiplayer games in the web browser (e.g. Curve Fever)
- Web analytics service (e.g. Google Analytics)
- Real-time collaboration software (e.g. Google Docs, Trello)

Real-time web, https://en.wikipedia.org/w/index.php?title=Real-time_web&oldid=716732622 (last visited May 14, 2016).

Some approaches for getting “real-time”

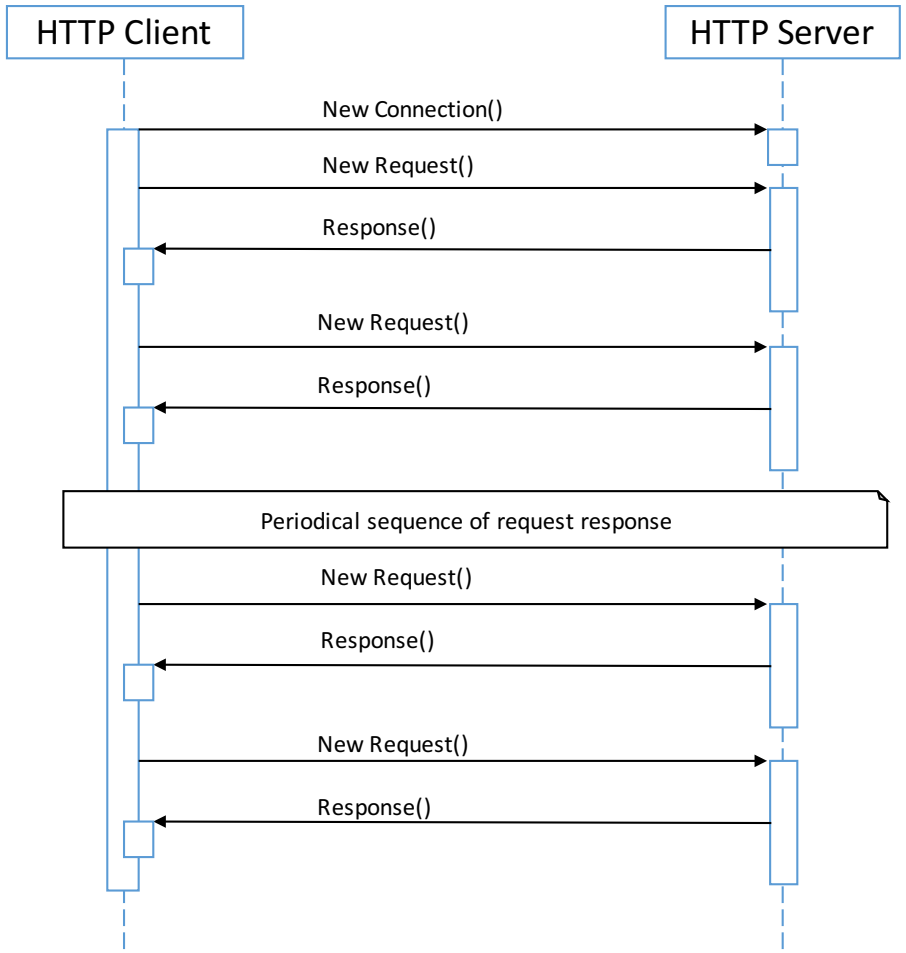
Short-Polling

Automating process of getting new information using refresh intervals like JavaScript *setInterval()*

→ Checking updates every n seconds

→ Conversations like this are resource intensive and inefficient! Take into account that each request/response totaled 871 bytes, just to find out nothing happens.

→ The data could change faster than the n defined seconds.



© sebis

Some approaches for getting “real-time”

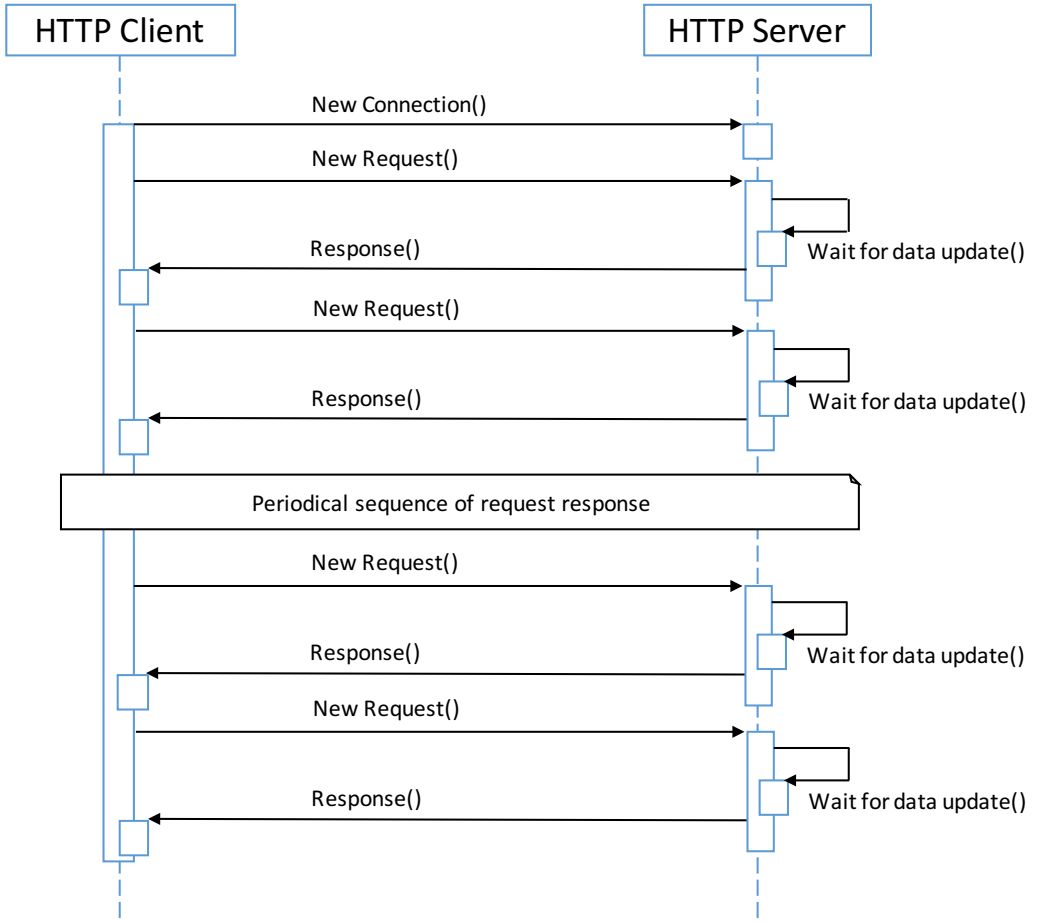
Long-Polling

Next step in the evolutionary chain of real-time web applications. Opening an HTTP request for a set period of time to listen for a server response.

→ New data available, the server will send it and close the request. Otherwise, request is closed after interval limit is reached and new one will be opened

→ More efficient then short-polling

→ Bidirectional communication will lead to double the resource: One for server-to-client message and one for client-to-server message.



Some approaches for getting “real-time”

HTTP-Streaming

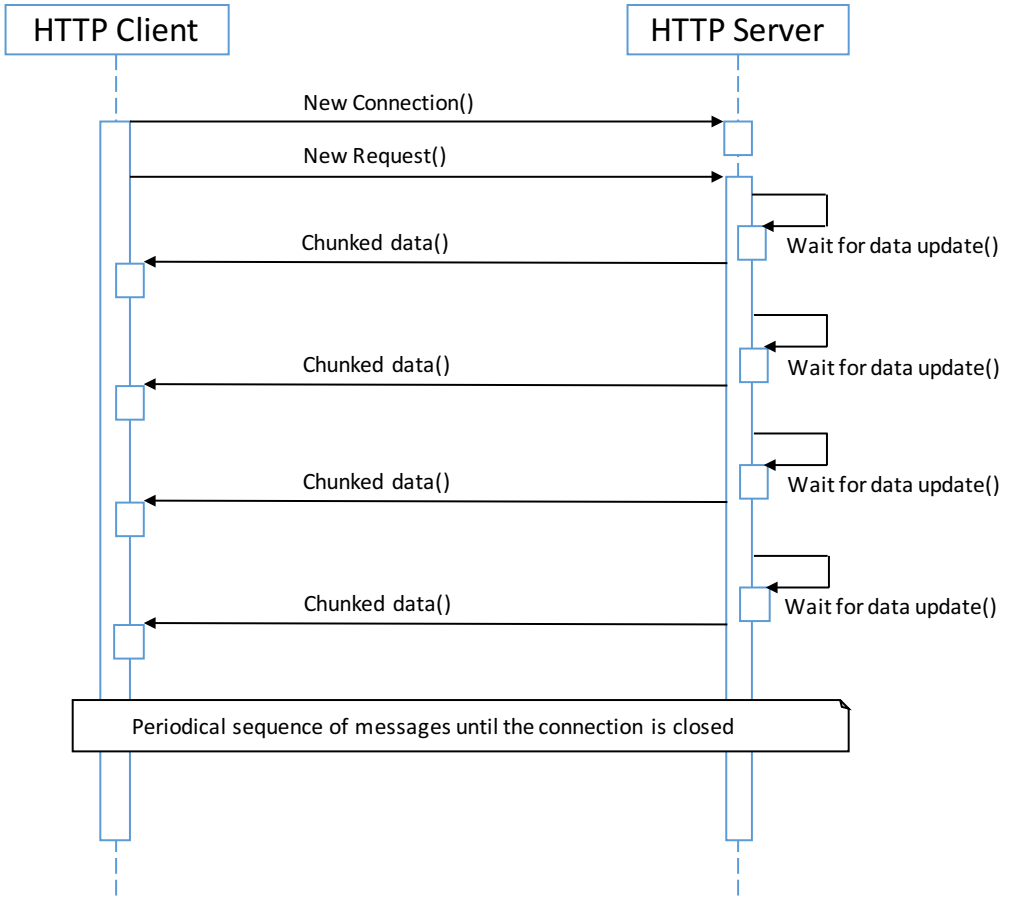
Similar to Long-Polling, except connection isn’t closed when new data is available or at a given interval.

→ Connection between client and server is persisted

→ Ensures that client and server is always in sync

→ However, still suffering from an inability to offer bidirectional communication without executing new request

→ Main Limitation: Inconsistencies of how http-streaming is achieved within different browsers.



© sebis

Some approaches for getting “real-time”

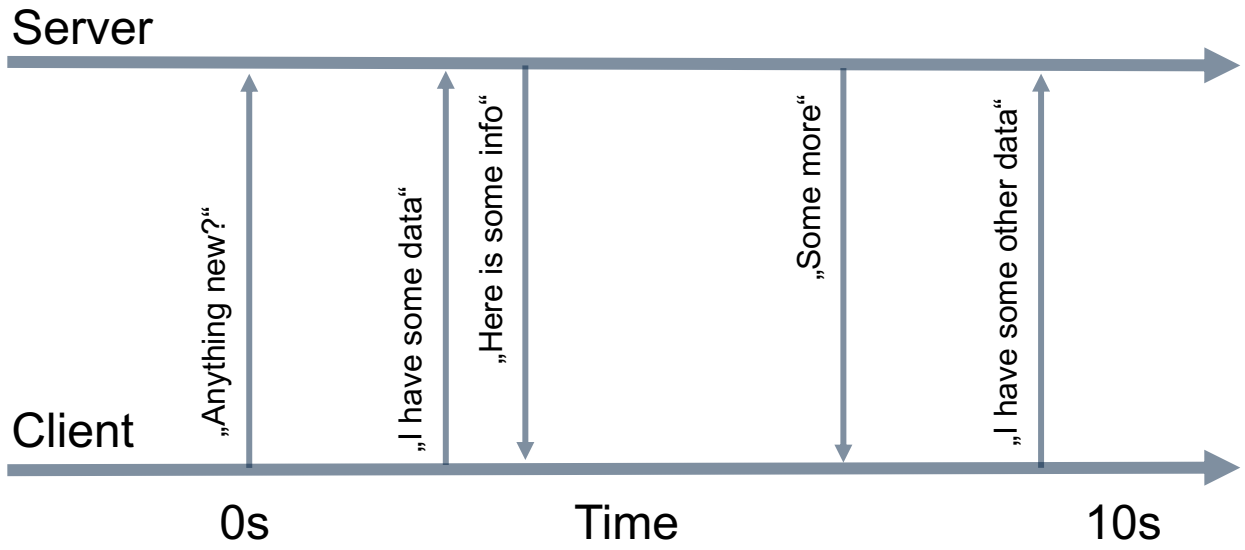
WebSockets

Standardized way for achieving client server bidirectional real-time and cross-domain communication over a single connection.

→ WebSocket specification is part of HTML5

→ Communication overhead is reduced as only one single TCP connection is used

→ The exchange between client and server is established via a *handshake*



Some approaches for getting “real-time”

WebSockets: Handshake

A bidirectional communication with the server is established by opening an HTTP request and asking to “upgrade” the connection to the WebSocket protocol:

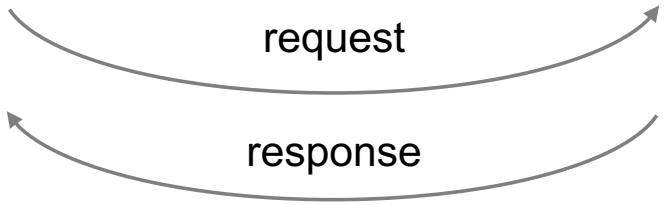
Client Header

GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13

Upgrade connection

Server Header

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat



Realtime web frameworks

Hosted real-time services

- MeteorJS
- Fanout
- Pubnub
- Pusher
- Realtime.co
- Streamdata.io
- Tambur.io
- Firebase

Open source real-time solutions

- Socket.io
- Deepstream.io
- Faye
- SockJS
- SocketCluster

Real-time databases for web apps

- RethinkDB
- Firebase

…and many more

© sebis

Socket.io sample code

Server side (/routes/socket.js)

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

exports.initialize = function (server) {
    io = io.listen(server); //start listening on server
    //new client has opened a connection

    io.sockets.on("connection", function (socket) {
        //send welcome message on the socket
        socket.send(JSON.stringify({
            type: 'serverMessage',
            message: 'Welcome to Socket.io real-time web service'
        }));

        //The „message“ handler is received
        socket.on('message', function (message) {
            message = JSON.parse(message);
            if (message.type == "userMessage") {
                //broadcast the message to all connected clients
                socket.broadcast.send(JSON.stringify(message));
            }
        });

        //Another event handler is received
        socket.on('anything', function (data) {
            //Do something in this event
        });
    });
};
```

Import Socket.io module

Export method **initialize** which can be called from the main application module (app.js).

The first event that our server will receive is a new connection from a new client.

The Method will **send** the message on the socket, which will be triggering the **message** event on the client

After **message** handler is received, we forward the message to all connected clients.

© sebis

Socket.io sample code

Server side (/app.js)

```
//start server on specified port
var server = http.createServer(app).listen(app.get('port'), function () {
    console.log("Express server listening on port " + app.get('port'));
});

//call socket module's initialize method
//passing this server as a parameter
require('./routes/sockets.js').initialize(server);
```

After we have modified the HTTP server, we can call the exported socket module's **initialize** method, passing this server as a parameter to it.

Socket.io sample code

Client side (/public/client.js)

```
//connect to server
var socket = io.connect('/');

//receive incoming message from server
socket.on('message', function (data) {
    data = JSON.parse(data);
    $('#messages').append('<div class="' + data.type + '">' +
        data.message + '</div>');
});

//bind click function to button
$(function () {
    $('#send').click(function () {
        var data = {
            message: $('#message').val(),
            type: 'userMessage'
        };

        //send message to server
        socket.send(JSON.stringify(data));
    });
});
```

This will send a connection request to the server and will also negotiate the actual transport protocol which will finally result in the **connection** event being triggered on the server side.

This will connect the event handler for the **message** event, enabling the client to receive incoming messages from the server.

Send JSON message to the server, triggering the **message** event on the server.

© sebis

Socket.io sample events

socket.on('connect', function () {}):

The connect event is emitted when the socket is connected successfully.

socket.on('connecting', function () {}):

The connecting event is emitted when the socket is attempting to connect with the server.

socket.on('disconnect', function () {}):

The disconnect event is emitted when the socket is disconnected.

socket.on('connect_failed', function () {}):

The connect_failed event is emitted when socket.io fails to establish a connection to the server and has no more transports to fall back to.

socket.on('error', function () {}):

The error event is emitted when an error occurs and it cannot be handled by the other event types.

socket.on('message', function (message, callback) {}):

The message event is emitted when a message sent by using socket.send is received. The message parameter is the sent message, and callback is an optional acknowledgment function.

Socket.io sample events

socket.on('anything', function(data, callback) {}):

The anything event can be any event except the reserved events. The data parameter represents the data, and callback can be used to send a reply.

socket.on('reconnect_failed', function () {}):

The reconnect_failed event is emitted when socket.io fails to reestablish a working connection after the connection was dropped.

socket.on('reconnect', function () {}):

The reconnect event is emitted when socket.io is successfully reconnected to the server.

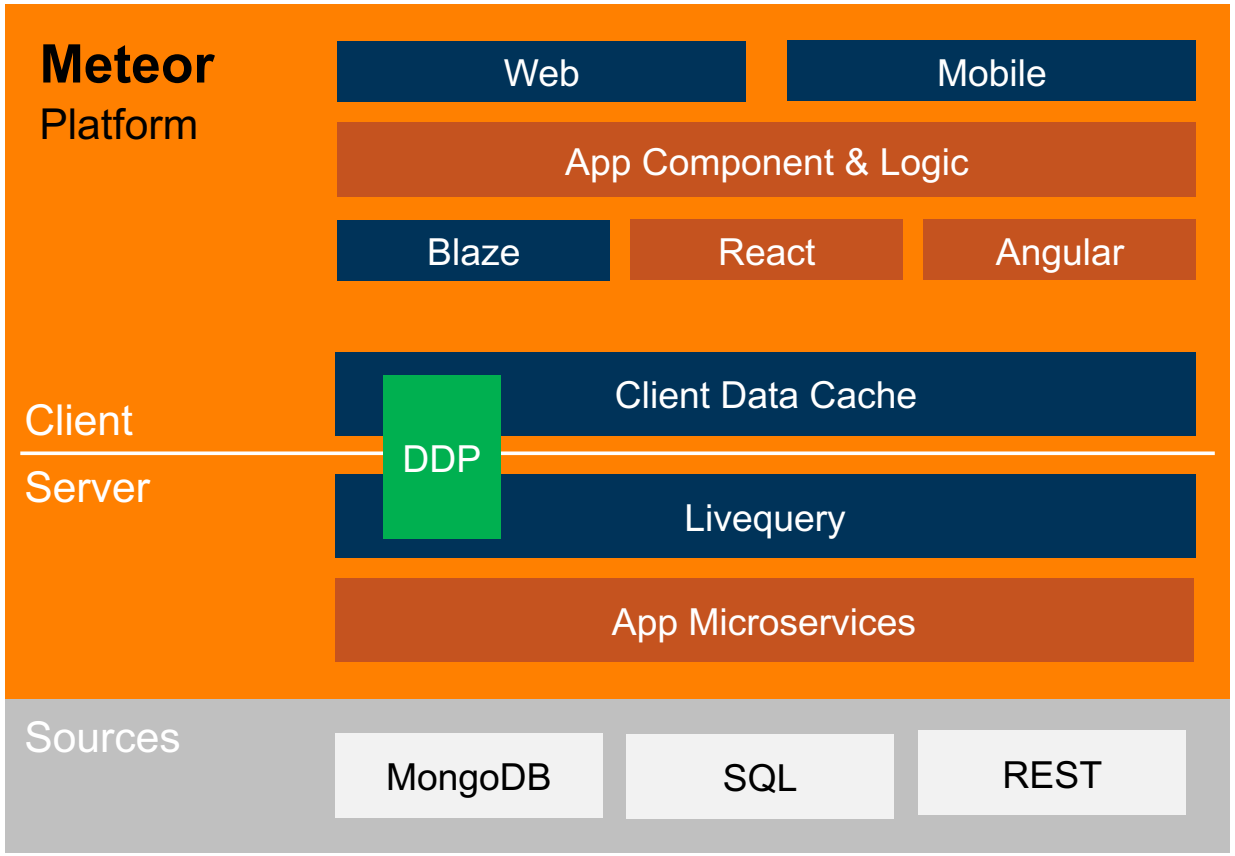
socket.on('reconnecting', function () {}):

The reconnecting event is emitted when the socket is attempting to reconnect with the server.

MeteorJS

A full-stack, open source platform for building web and mobile apps in JavaScript

- Reactive single-page applications
- One codebase for all platforms
- Built on top of Node.js



© sebis

Meteor core principles

1. **One language:** Meteor allows to develop with JavaScript in all environments: application server, web browser, and mobile device.
2. **Data on the wire:** The server only sends raw data to the client but no HTML. The actual HTML representation of the data is composed and rendered on the client.
3. **Embraces the ecosystem**: The JavaScript community is large and active. Meteor integrates various other open source projects and doesn't reinvent the wheel.
4. **Reactive:** Seamlessly reflect the true state of the world with minimal development effort.

Reactive source & reactive computations

Reactive sources	Reactive computations
 **Objects** that **notify** their context when something **inside changes**	A **piece of code** that **runs whenever** a **reactive data source inside** that piece of code **changes**

Example:

```
// however, the actual Meteor
// implementation looks a bit different
Session.set('x', 823);
Session.set('y', 514);
var z;

// method re-runs as soon as one of the session //
variables change
Tracker.autorun(function(){
    z = Session.get('x') +
        Session.get('y');
    alert(z);
});
Session.set('y', 1); // callback triggered
```

Session variables are *reactive sources*

Tracker autorun will re-run as soon as session variable y or z changes

Tracker autorun defines the scope for a *reactive computation*

© sebis

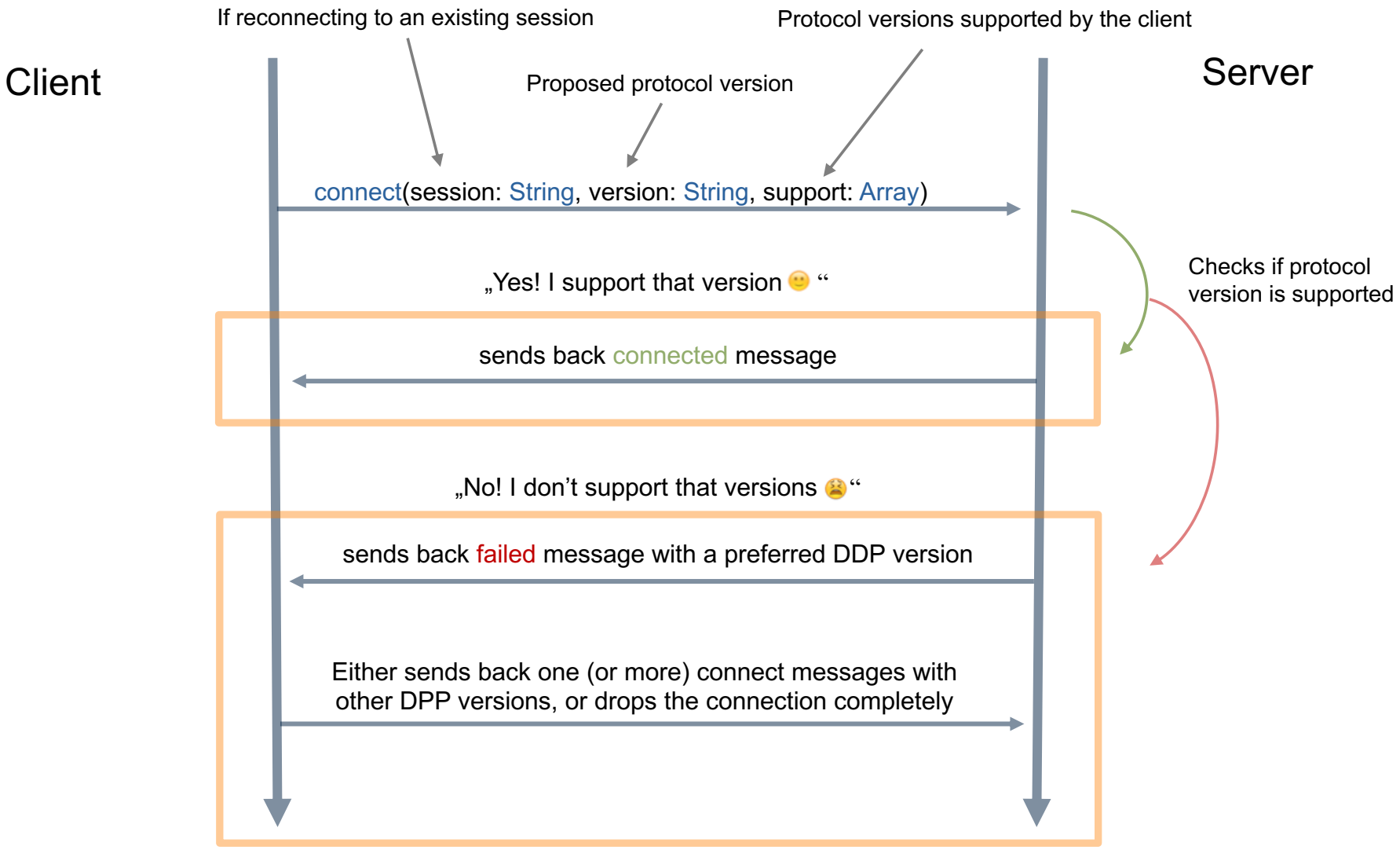
Meteor core components

DDP (Distributed Data Protocol)

- Protocol created by the Meteor team to exchange data between the client and the server.
- Very minimalistic: Only **two operations** are supported:
  1. **Remote procedure calls** (RPC) by the client to the server.
  2. **Publish-subscribe messaging**, i.e. clients subscribe to a set of documents and the server keeps the client up-to-date about the current state and content of the documents.
- All **DDP messages** are simple **JSON objects**. Each JSON object must contain a ***msg*** field where the type of the message is specified. If the JSON object contains unknown fields, the client, as well as the server, must ignore them.
- Uses sockets as underlying transport layer. Meteor may **use WebSockets** or **SockJS** which can emulate WebSockets if they are not available.

© sebis

DDP connection



© sebis

Meteor core components

Blaze

- Meteor Blaze is a library for creating live-updating user interfaces.
- Blaze fulfills the same purpose as Angular (UI Framework), Backbone, Ember, React, Polymer, or Knockout
- Blaze templates are reactive, i.e. automatically update on changes
- Spacebars templates with special tags delimited by curly braces: {{ }}
- Feel more comfortable with the Angular UI Framework? Meteor does not force you to use Blaze! You can easily switch to Angular if you want by using Angular Meteor (http://www.angular-meteor.com/)

Blaze templates

```
<!-- myApp.html -->

<html>
{{>friendsList}}
</html>
```

```
<!-- Template definition -->
<template name='friendsList'>
   <ul>
      {{#each friends}}
         <li class='{{#if selected}}selected{{/if}}'>
            {{firstName}} {{lastName}}
         </li>
      {{/each}}
  </ul>
</template>
```

Import the defined template into the HTML frame of our app.

Templates are defined HTML snippets that are enclosed in <template> tags. Each template can be assigned with a specific name.

Iterate over a collection named *friends*. Usually, this is a list of objects.

selected, *firstName* and *lastName* are attributes of a single *friend* object.

© sebis

Controller

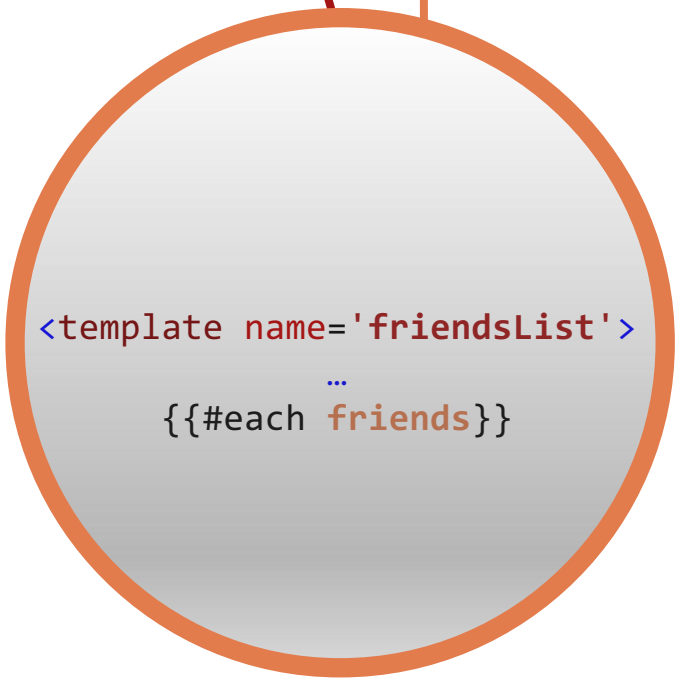
```
// myApp.js

Friends = new Meteor.Collection('friends');

Template.friendList.friends = function () {
    return Friends.find();
};
```

Select the collection *named* friends from the apps' database.

Bind the variable *friends* in the template named *friendsList* with the documents found in the selected collection.

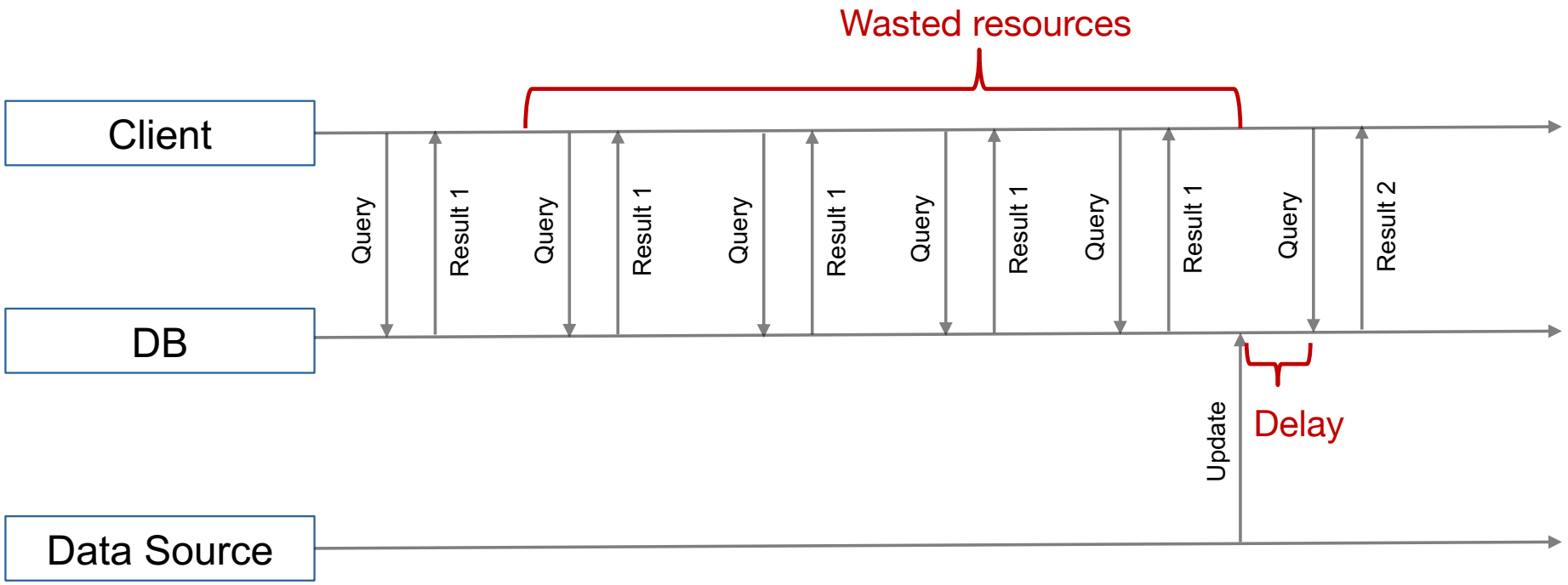


Meteor core components

LiveQuery

- Family of live database connectors to perform "live queries" against the database
- LiveQuery informs the user about any CRUD operations performed on the database
- Enables real-time polling of data changes and solves problems that occur by “traditional” pulling of real-time data

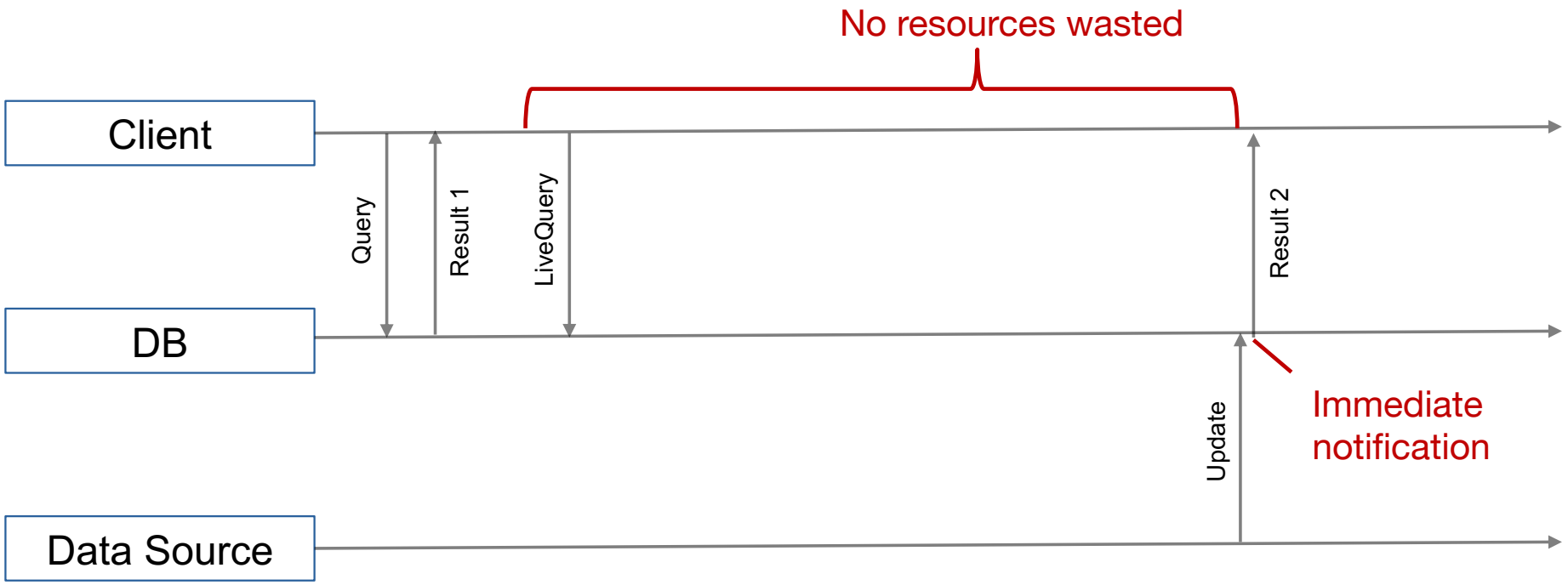
Traditional query polling wastes resources



LiveQuery benefits

LiveQuery solves the **2 main problems** that a traditional polling approach has:

1. Waste of resources
2. Delay time when data is updated



© sebis

LiveQuery change detection

How does the LiveQuery change detection work?

- LiveQuery does only work properly if it receives a live feed of occurring database changes.
- There are several options to achieve such a live feed:
  - Set up database triggers that fire when some pre-defined event happen, e.g. the insertion of a new data entry. This is the default option for SQL databases.
  - Not all databases support triggers, e.g. MongoDB. In that case, LiveQuery can connect to the database and pretend to be a replication slave. The idea behind this approach is that LiveQuery then consumes the replication log and use it a as live state feed.
  - Since most databases support some kind of replication, this is a widely applicable approach.
  - However: If the user does not have sufficient rights to read the replication log, or to set up some kind of triggers, then the only option is to periodically poll for changes. In that case, we are basically back to the traditional polling approach.
- Visit https://www.meteor.com/livequery for a detailed documentation

© sebis

Meteor core components

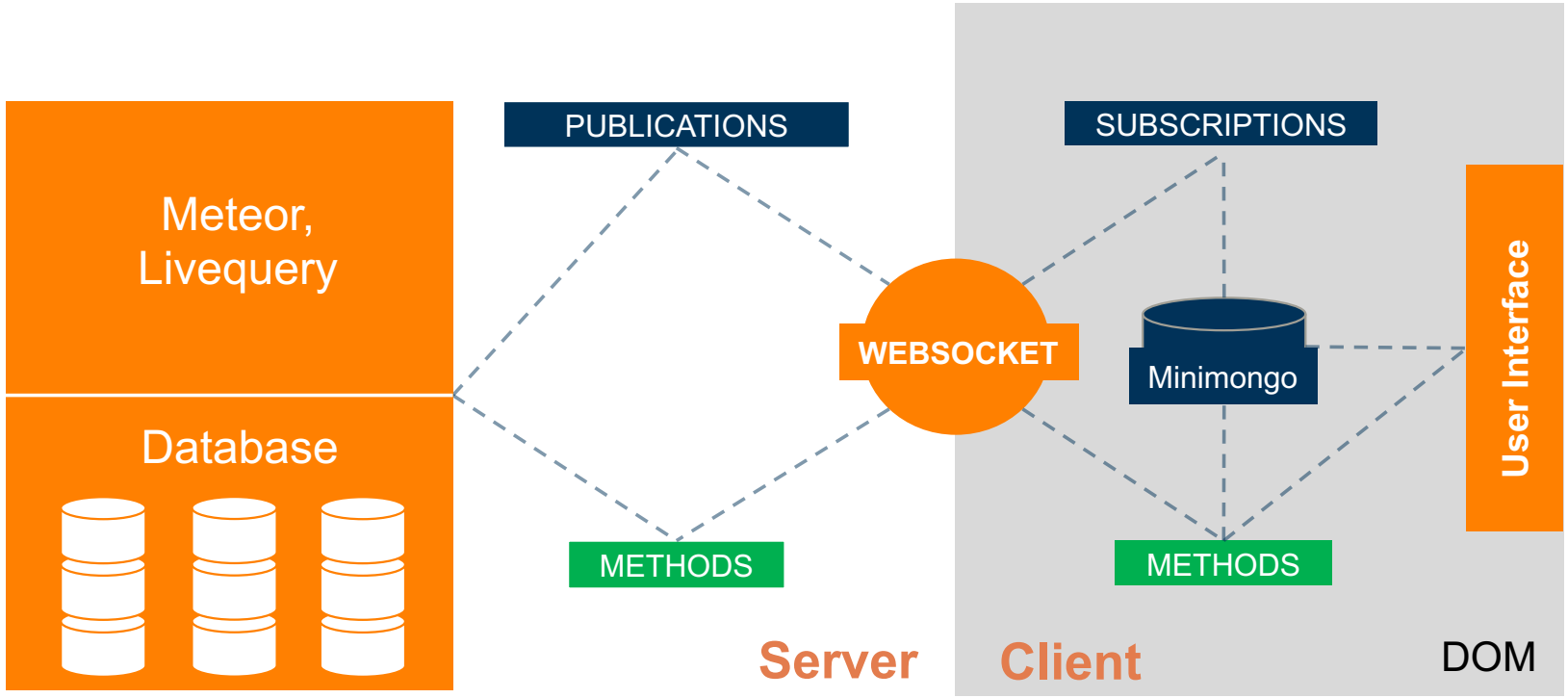
Minimongo

- A client-side implementation of MongoDB which supports basic queries (not all!)
- Minimongo uses either IndexedDb, WebSQL, Local storage or is in-memory only
- It is like a MongoDB emulator that runs inside the web browser
- Acts as a datastore on the frontend and allows the application to make database queries on the client side
- Minimongo is required to store incoming data from the server
- Features: Live Queries, tracker-ready and changes tracking
- It’s open source

© sebis

Meteor data flow

- The core concept of exchanging data between the server and the client in Meteor is the DDP and the publish and subscribe pattern
- Meteor.publish and Meteor.subscribe are the two main methods to orchestrate the data flow
- Minimongo is used to store the data retrieved by the subscriptions and to run queries on the currently subscribed to documents



© sebis

Outline

7. Advanced topics in web application engineering

- Transcompilers for the web
- Real-time web applications
- Hybrid web applications

Mobile apps with Meteor and Cordova

Cordova

- A framework to build mobile apps with **HTML**, **CSS** and **JavaScript**
- Create iOS, Android and WindowsPhone apps with a **single code base**
- Access **native functionality** like the camera or the microphone **via JavaScript**
- Native functionality can be extended via plugins
- Fast development cycles
- Ideal to create app prototypes and proof of concepts
- App runs in a web view which is embedded in a native app wrapper
- Bundles all assets (HTML, CSS, JS) directly into the app. This makes the app very fast and responsive
- Most used solution for building modern hybrid apps

Mobile apps with Meteor and Cordova

Meteor Mobile

- Meteor comes with a built-in integration with Apache Cordova
- **Development and concepts** are **the same** as for **any** other **platform** – JavaScript all the way
- Native Cordova plugins can be used to get access to native APIs such as contacts and the exact geolocation
- Meteor supports hot code push → Update your app instantly without re-uploading it to the App/Play-Store

Requirements

iOS	Android
You need a Mac with installed xCode to test your app on a real device or in the simulator.	For Android, you need a properly installed Java Runtime Environment and the Android SDK
Even if you have one code base, you should not release anything not tested on a actual device!	Meteor will handle everything on first build and installs missing libraries

Mobile apps with Meteor and Cordova

Set up new platforms on with Meteor

```
[Terminal // CMD]
~ Patrick$ cd /MeteorProjectPath

# Add and remove platforms
meteor add-platform ios #add ios
meteor add-platform android #add android
meteor remove-platform ios android #remove ios and android
meteor list-platform #shows a list of all installed platforms

# Run the app
meteor run ios #run app in simulator
meteor run ios-device #run app on actual device
meteor run android #run app in emulator
meteor run android-dive #run app on device

# Add native plugins
meteor add cordova:cordova-plugin-camera@1.2.0
# https://cordova.apache.org/plugins/
```