Push

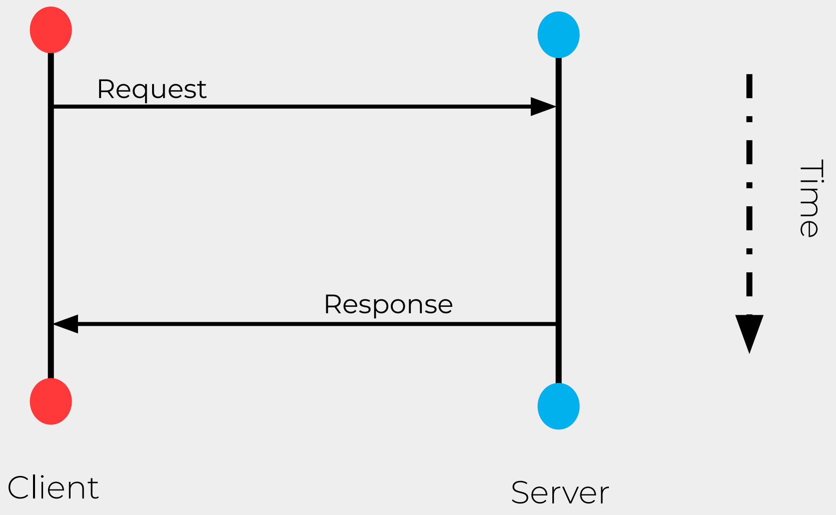
Writing multithreaded apps and updating UIs asynchronously in Vaadin 14

Suggested trainings before this one:

- Application Lifecycle

Recap: Servlet applications

A Servlet application takes in **HTTP Requests** and returns a **HTTP Response** for each. The communication is **synchronous**.

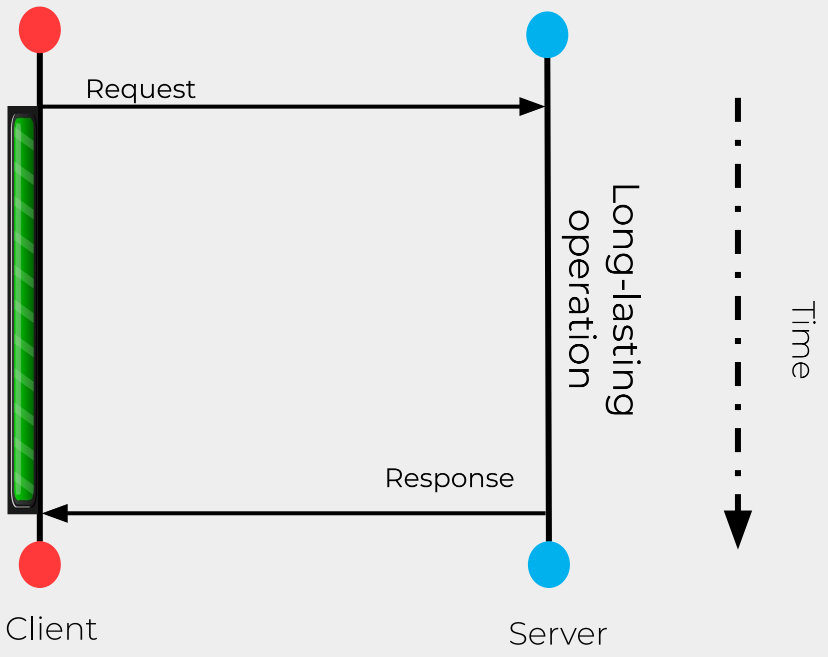


Recap: Servlet applications

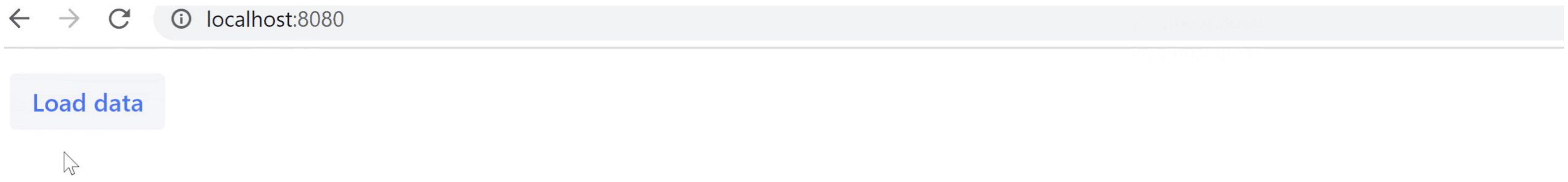
The synchronous application has two downsides:

1) After a request has been sent, the UI will remain unresponsive until the response arrives

2) Any updates to the UI must come as a reaction to some event on the client



Updating a UI synchronously

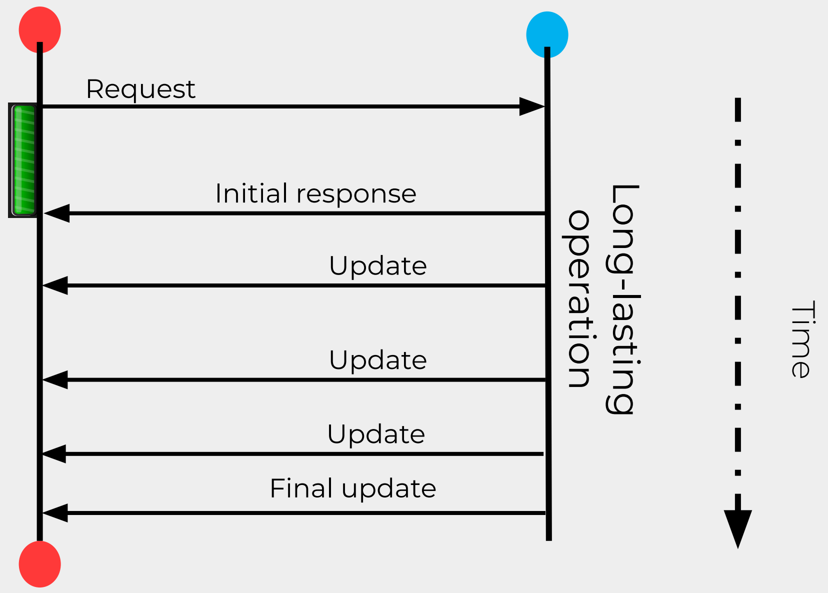


The only feedback the user gets is the loading indicator

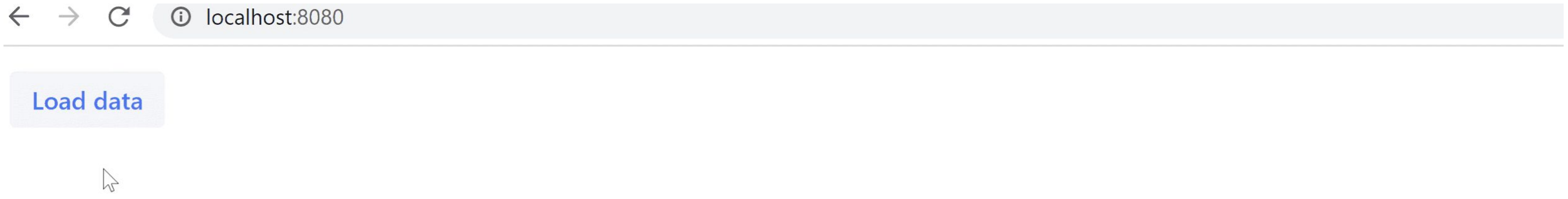
Asynchronous updates

Both of the problems can be solved with **asynchronous updates**.

In other words: as well as reacting to requests, the server can send updates to the client whenever it needs to.



Updating a UI **asynchronously**

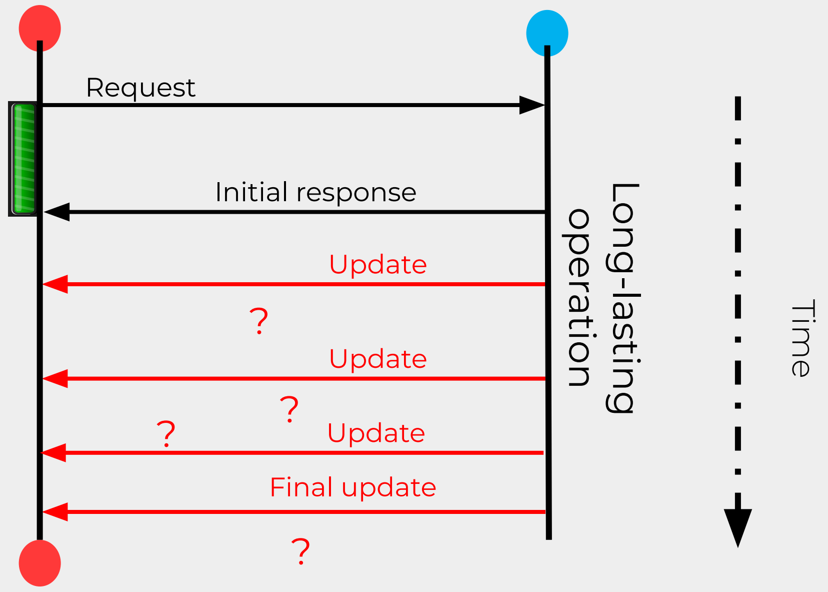


You can add components to the UI to indicate that something is meant to take a while

Asynchronous updates

But HTTP is a synchronous protocol - there can't be responses without requests and the server can't initiate a HTTP request to the browser.

How can we resolve this situation?



Multithreading

You should avoid creating Threads directly. Modern Java application servers do a great job in managing threads - use ExecutorService or ManagedExecutorService instead:

```
ExecutorService executor = Executors.newCachedThreadPool();

executor.submit(()->{
   // This will take some seconds...
   List<Person> personList = personService.getEmployees();
   personGrid.setItems(personList);
   hideLoadingText();
});
```

Spring users can use TaskExecutor for similar capabilities.

Multithreading

Or use CompletableFuture if there is a return value and you need more control over the task

```
ExecutorService executor = Executors.newCachedThreadPool();

CompletableFuture<List<Person>> future =
       CompletableFuture.supplyAsync(personService::getEmployees, executor);
```

Useful methods:

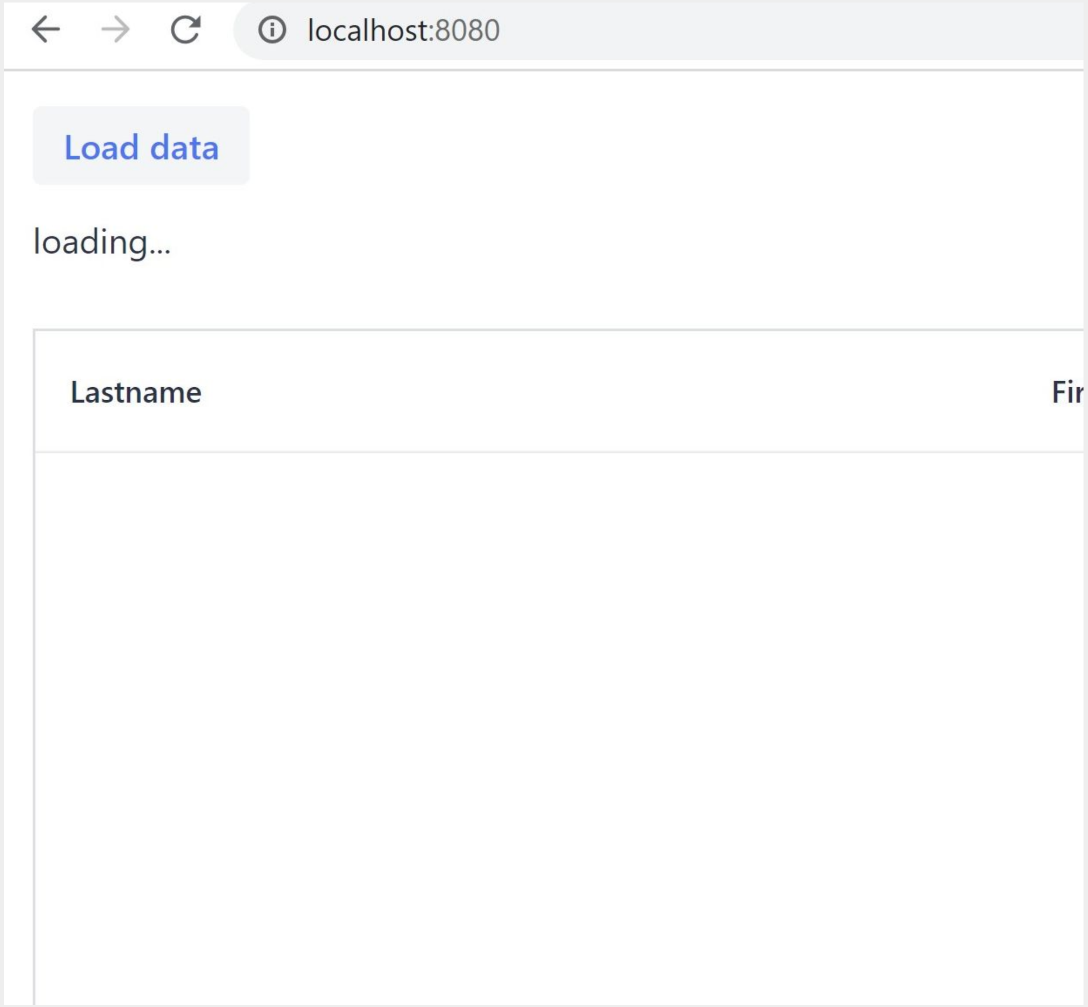
```
future.thenAccept(List<Person> persons -> { /* do something with return value */ })

future.isDone()

future.cancel(boolean useInterruptIfNeeded)
```

But...

Multithreading

...even if you do this, the data never shows up. Instead, you might see an exception in the log

```
Exception in thread "Thread-15"
java.lang.IllegalStateException: Cannot access
state in VaadinSession or UI without locking the
session.
```



loading...

Lock the UI

Use the UI::access callback to lock the UI when doing changes to the UI states from other threads.

UI::access takes in a Command (serializable Runnable) as parameter

You should only put the UI-updating code into the UI::access method

```
executor.submit(()->{
   // do the heavy work before calling ui.access
   List<Person> personList = personService.getEmployees();
   ui.access(()->{
       grid.setItems(personList);
       hideLoadingText();
   });

});
```

Where to get the UI instance?

Using `UI.getCurrent()` inside a non-request handling thread **won't work**, because UI instances are stored in a `ThreadLocal Map`.

You can get the UI instance from a component by calling getUI() method outside the thread.

```
Button button = new Button("Load data",
       event -> {
           add(personGrid);
           showLoadingText();
           // UI instance is retrieved from the getUI() method before starting a thread
          UI ui = getUI().get();
           executor.submit(()->{
              // do the heavy work before calling ui.access
              List<Person> personList = personService.getEmployees();
              ui.access(()->{
                 grid.setItems(personList);
                 hideLoadingText();
              });

       });
);
```

What does UI::access() do?

The method `access` provides exclusive access to the UI from outside a request handling (a different Thread)

The given command is executed while holding the session lock to ensure exclusive access to this UI

If the lock is available, the command is run right away

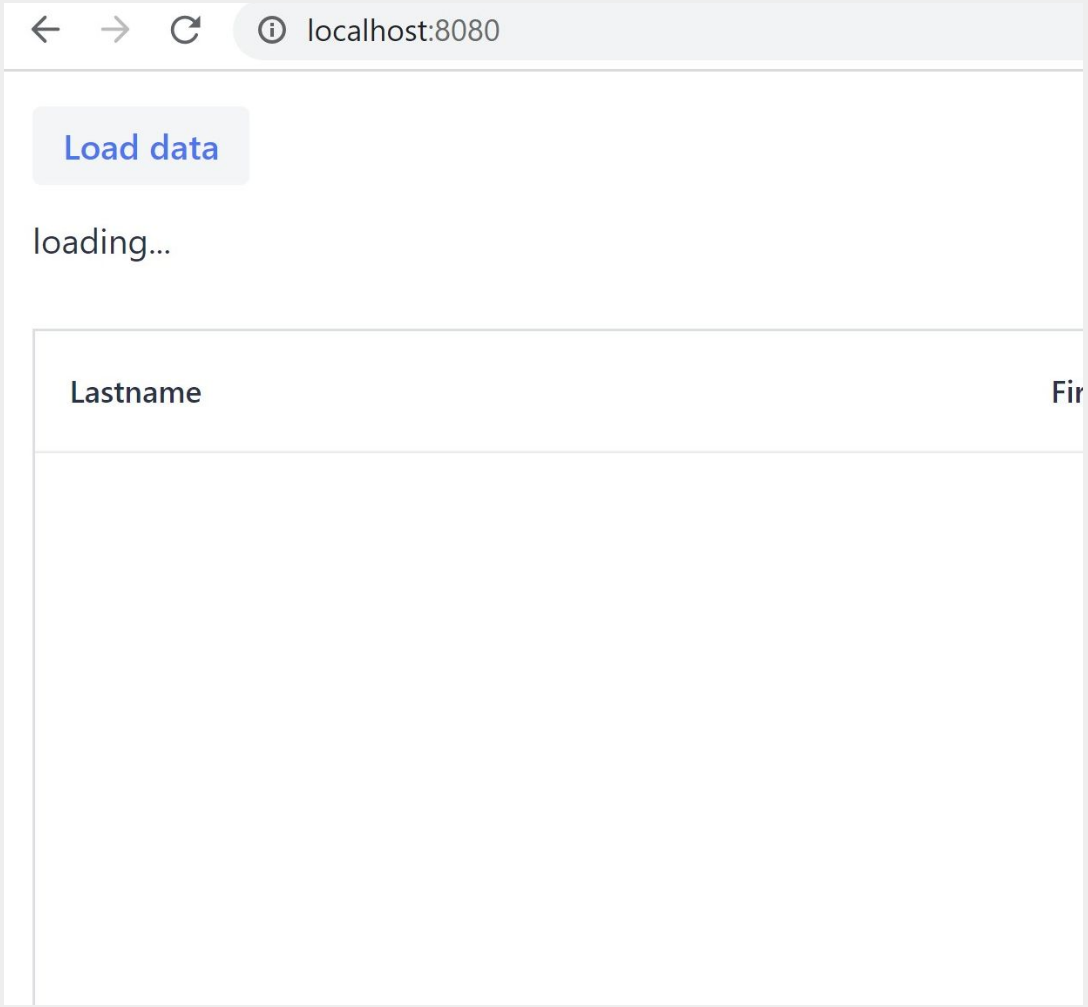
If the lock is not available, the job is left in a queue, eventually run by the one who releases the lock.

Hence, keep updates done in UI::access as atomic as possible in order to avoid blocking the UI.

Still not working

Multithreading + UI::access ≠ Success

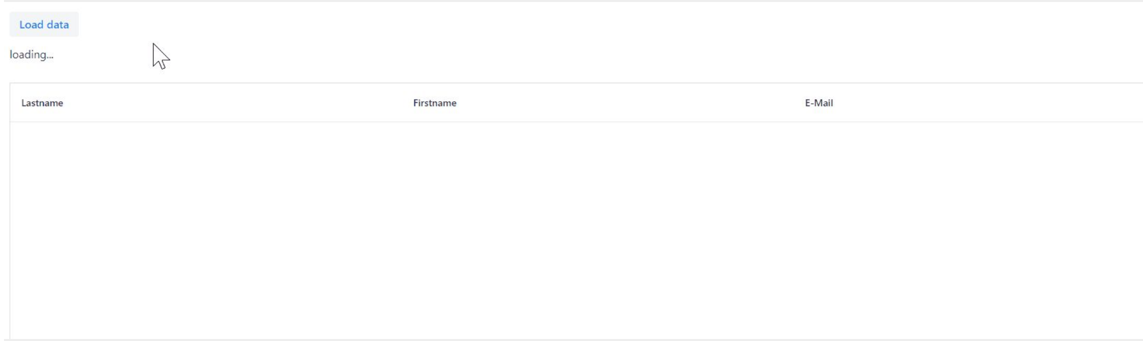
No more exceptions, but the data still doesn’t show up



Observation

Wait for a while, click the button again.

Suddenly the data shows up.



Why?

Remember: Because of the nature of HTTP, the communication always starts on the client-side.

The server has no way of initiating a conversation with the client.

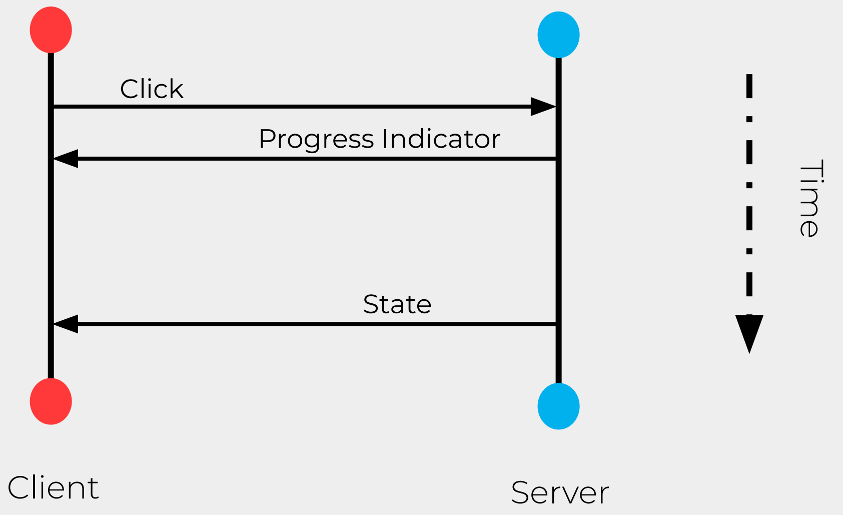
That’s why updating the UI from a background thread doesn’t work alone.

Solution: Push

Best choice if supported by browser & server

Minimal amount of traffic

Client-side always receives message at the "right" time



@Push

To enable push, only need to add @Push annotation to the root layout

```
@Route("")
@Push
public class MainView extends VerticalLayout {
   public MainView() {
       ExecutorService executor = Executors.newCachedThreadPool();
       Button button = new Button("Load data",
       event -> {
           add(personGrid);
           showLoadingText();
           UI ui = getUI().get();
           executor.submit(()->{
              List<Person> personList = personService.getEmployees();
              ui.access(()->{
                 grid.setItems(personList);
                 hideLoadingText();
              });
       });
       );
       add(button);
   }
}
```

Muitithreading + UI::access + @Push = Success!

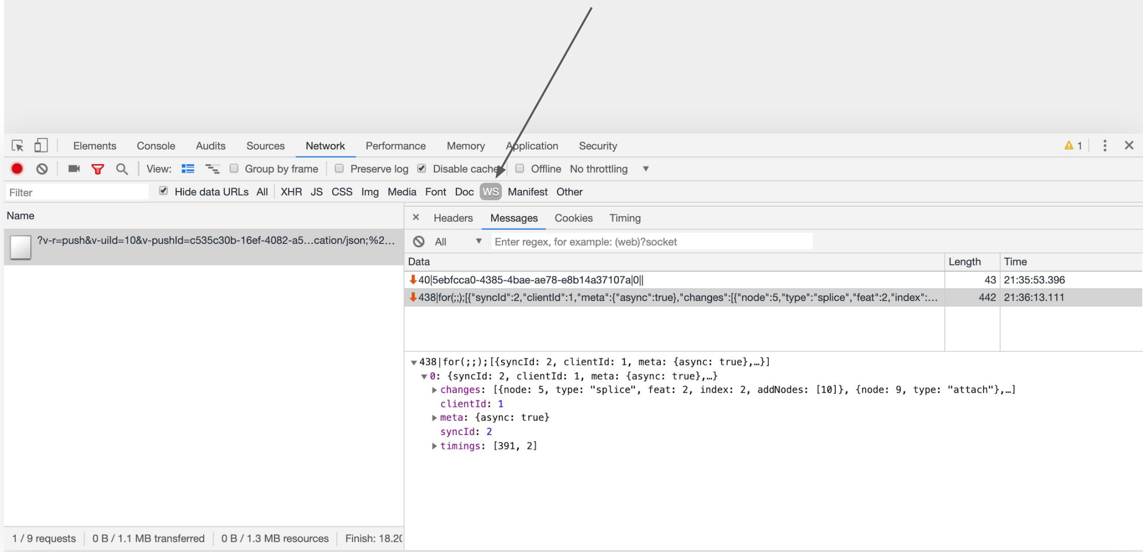
@Push transport options

	Normal client to server requests	Asynchronous server to client updates
No push	HTTP	N/A
`WEBSOCKET`	Websocket	Websocket
`WEBSOCKET_XHR` `(Default in Vaadin)`	HTTP	Websocket
`LONG_POLLING` `(Fallback in Vaadin)`	HTTP	Indefinite HTTP request

Websocket traffic

?v-r=push means it's a push request

WS means WebSocket, you can observe WebSocket traffic there.



Name

?v-r=push&v-uild=10&v-pushId=c535c30b-16ef-4082-a5…cation/json;%2…

Data	Length	Time
40|5ebfcca0-4385-4bae-ae78-e8b14a37107a|0||	43	21:35:53.396
438|for(;;);[{"syncId":2,"clientId":1,"meta":{"async":true},"changes":[{"node":5,"type":"splice","feat":2,"index":…	442	21:36:13.111

```
▼438|for(;;);[{syncId: 2, clientId: 1, meta: {async: true},…}]
  ▼0: {syncId: 2, clientId: 1, meta: {async: true},…}
    ▶changes: [{node: 5, type: "splice", feat: 2, index: 2, addNodes: [10]}, {node: 9, type: "attach"},…]
     clientId: 1
    ▶meta: {async: true}
     syncId: 2
    ▶timings: [391, 2]
```

1 / 9 requests | 0 B / 1.1 MB transferred | 0 B / 1.3 MB resources | Finish: 18.2

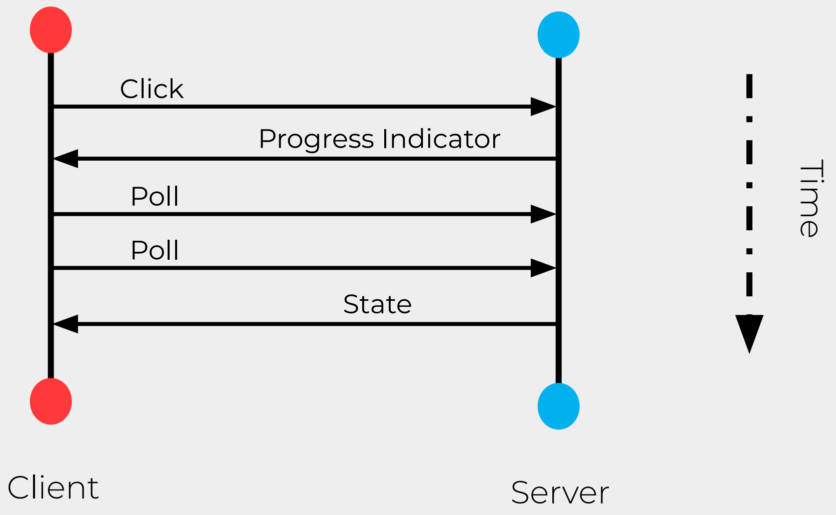
Workaround: Polling

```
UI::setPollInterval(interval)
+
UI::addPollListener(listener)
```

You can configure Vaadin to create polling requests every n milliseconds and react to those requests with a poll listener.

This generates extra traffic, which can be heavy with multiple clients.

It is difficult to find the correct polling interval - there's no one right value for all use cases. Too fast polling throttles the network, too slow doesn't react quickly enough.



Exercise

Feedback

bit.ly/vaadin-training