POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Desenvolvimento de Aplicações Distribuídas

Communication

Communication between distributed components

Marco Monteiro and Ricardo Gomes

Summary

1. Communication Essentials
2. Communication Models and Technologies
3. REST API
4. REST API with Laravel
5. WebSockets
6. WebSockets with Socket.IO

Communication Essentials

Stateless and stateful

Unidirectional and bidirectional

Message formats and serialization

JSON

Stateless vs Stateful Communication

- Stateless
  - No server session; context sent with every request
  - Request/response is the dominant stateless pattern
  - Scales horizontally; simpler caching and recovery
  - Examples: REST, GraphQL queries, gRPC unary, Webhooks
- Stateful
  - Server/connection maintains interaction or session context
  - Enables streaming, presence, low-latency coordination
  - Examples: WebSockets, GraphQL subscriptions, gRPC streaming, message brokers

Unidirectional vs Bidirectional Models

- Unidirectional
  - One direction per interaction or stream; sender → receiver
  - Patterns: request/response; server streaming; client streaming
  - Simple semantics; cache-friendly; resilient intermediaries
  - Examples: REST, GraphQL queries, gRPC unary, SSE, Webhooks
- Bidirectional
  - Full-duplex communication on a persistent channel
  - Both peers can send anytime; interleaving allowed
  - Ideal for real-time coordination and low-latency feedback
  - Examples: WebSockets, gRPC bidirectional streaming, GraphQL subscriptions

Message Formats & Serialization

- Serialization converts objects into a message format; deserialization reconstructs structure.
- Common message formats
  - JSON, XML (text); Protobuf, Avro (binary)
  - Typical use: JSON (REST/GraphQL/WebSockets); XML (SOAP); Protobuf (gRPC); Avro (Kafka)
- Schemas define fields, types, and compatibility rules.
  - JSON Schema; XML Schema (XSD); Protobuf .proto; Avro Schema/IDL
- Service contracts advertise capabilities
  - OpenAPI (REST); GraphQL SDL (GraphQL); Protobuf .proto (gRPC)

JSON: Overview & Syntax

- JSON = JavaScript Object Notation
- Lightweight text data format; widely adopted
- Human-readable; easy to generate and parse
- Default choice for web APIs and client messaging
- Language-agnostic, but:
  - derived from JavaScript object literal syntax
  - maps to JavaScript objects, arrays, and primitives
- Strict syntax rules:
  - double-quoted (") strings and property names - no (') or (`)
  - no comments
  - no trailing commas
  - numbers are IEEE-754 doubles

JSON: Syntax

- Object properties are represented in `name : value` pairs
  - Property names must be double-quoted strings

```
"name" : value
```

- Data is separated by commas (no trailing comma):

```
{ "name1" : value1, "name2" : value2 }
```

- Curly braces hold objects (with a set of properties):

```
{ "name1" : value1, "name2" : [1, "a"] }
```

- Square brackets hold arrays (with a set of elements):

```
[ 12, "abc", true, {"name": value} ]
```

JSON: Types and Values

- JSON defines six value types: number, string, boolean, null, array and object

Value type	Description and examples

Number	`"grade": 12`
	`"speed": 435.3`

String	`"name": "John Doe"`
Boolean	`"hasGrade": false`
Null	`"finalGrade": null`
Array	Ordered collection of values (not pairs) `[ 12, "abc", true, {"n1": 12}, null, 2.1 ]`
Object	Unordered collection of pairs `{ "n1": 12, "n2": true, "n3": [2,"a",3]}`

JSON: Example

- Example of a JSON string:

```json
{
  "id": 42,
  "type": "event",
  "ok": true,
  "customers": ["john_doe", "jane_smith"],
  "data": { "user": "alex", "games": [12, 23] },
  "payment_reference": null
}
```

Communication Models and Technologies

Representative Communication Models and Technologies in Distributed Applications

REST API

- Coined by Roy Fielding (2000); leverages HTTP and Web standards
- Resource-oriented HTTP interface for client–server communication
- Stateless interactions via unidirectional request/response; each request is self-contained
- Format-agnostic; JSON most common via negotiation
- Used for CRUD, web/mobile backends, and third-party integrations
- Ubiquitous on the web; dominant for public/internal APIs
- Request/response pairs with verbs, status codes, headers, caching

REST API Request/Response Pair

HTTP Request:

```
GET /api/v1/users/42
Accept: application/json
```

HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
{ "id": 42, "name": "Alex" }
```

GraphQL

- Created at Facebook (2012), GraphQL Foundation 2018
- Client-driven query language and runtime; GraphQL fetches exactly needed data
- Designed to reduce REST over/under-fetching and multiple round trips
- Stateless HTTP request/response; single endpoint; JSON responses
- Advantages over REST: fewer calls per view; flexible aggregation; typed schema
- Disadvantages over REST: harder caching/CDN; increased server complexity; cost limits
- Adoption: common for web/mobile frontends and app-specific backends; not universal

GraphQL Request/Response Pair

HTTP Request:

```
POST /graphql
Content-Type: application/json
{ "query": "query { user(id: 42) { id name } }" }
```

HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
{ "id": 42, "name": "Alex" }
```

WebSockets

- RFC 6455 (2011); Web Standard (W3C WebSocket API); universal browser support
- Bidirectional, full-duplex messaging over a persistent connection
- Stateful channel; server tracks connections, rooms, subscriptions
- Low-latency push; ideal for real-time coordination and presence
- Used for chat, live dashboards, multiplayer, collaborative editing
- Implementation: event emit/on, rooms/channels, heartbeats, reconnection

Communication is bidirectional:

Client --> Server

```
socket.emit("join", {"game": 42})
```

Server --> Client

```
socket.emit("joined", {"game": 42})
```

Server --> Room Broadcast

```
socket.to("game:42")
  .emit("update", {"game": gameStatus})
```

gRPC

- Created at Google; open-sourced in 2015; HTTP/2-based RPC
- Binary Protobuf messages; compact serialization; strong typing
- Optimized for high-performance, low-latency service-to-service calls
- Four styles: unary; server streaming; client streaming; bidirectional
- Suited to microservices/internal APIs; small payloads, multiplexing, flow control
- Browsers require gRPC-Web via proxy; servers/mobile are first-class
- Adoption: common in modern backends; uncommon for public web APIs

Request:

```
rpc: UserService.GetUser
metadata: grpc-timeout: 1s
message: { user_id: 42 }          // Protobuf (binary)
```

Response:

```
status: OK  // gRPC status 0
message: { id: 42, name: "Alex" } // Protobuf (binary)
```

Message brokers

- Kafka (LinkedIn → Apache, 2011); RabbitMQ (AMQP, 2007); many others
- Purpose: decouple producers and consumers; buffer spikes; operate at different speeds
- Patterns: queues for work distribution; topics/logs for publish/subscribe fan-out
- Delivery semantics: at-least-once by default; ordering within a queue/partition
- Durability & replay: persisted logs, offsets and acknowledgments; centralized backpressure handling
- Adoption: widespread in enterprise and data pipelines; not a direct browser protocol

Publish:

```
topic: user.events        key: 42       headers: { "trace-id": "ab12-..." }
value:{ "type": "UserCreated", "userId": 42, "occurredAt": "2025-11-06T10:00:00Z" }
```

Subscribe:

```
consumer-group: billing-service
subscribe: user.events
on message (m):
  chargeAccount(m.value.userId)     // process event
  commit offset                     // mark as processed
```

REST API

Core Concepts

REST Architecture Basics

- REST = **Re**presentational **S**tate **T**ransfer - Architectural style for distributed systems on the Web
- Key abstraction in a REST architecture is the **Resource**
- Resources are nouns identified by URLs and manipulated via standard HTTP methods.

  "users"; "products"; "products/32"; "products/32/photo"; "today's weather in Leiria"
- Transfer representations of resource state (e.g., JSON)
- REST architectural constraints:

1. Client–Server
2. Stateless
3. Cacheable
4. Uniform Interface
   1. Resource identification
   2. Manipulation through representations
   3. Self-descriptive messages
   4. HATEOAS (Hypermedia as the Engine of Application State)
5. Layered System
6. Code-on-Demand (*optional*)

Uniform Interface: Resource Identification

- **Description**: Identify resources with stable, canonical URIs
- Nouns, plurals, hierarchy; avoid verbs/RPC shapes (Remote Procedure Calls Shapes)
- One stable URL per resource; HTTP method conveys the action
- Use query params for filter/sort/page, not identity
- Versioning explicit: /v1 (or version header)

Avoid:

```
createUser
getUser?id=42
```

✗

```
blockUser?id=42
doPayment
```

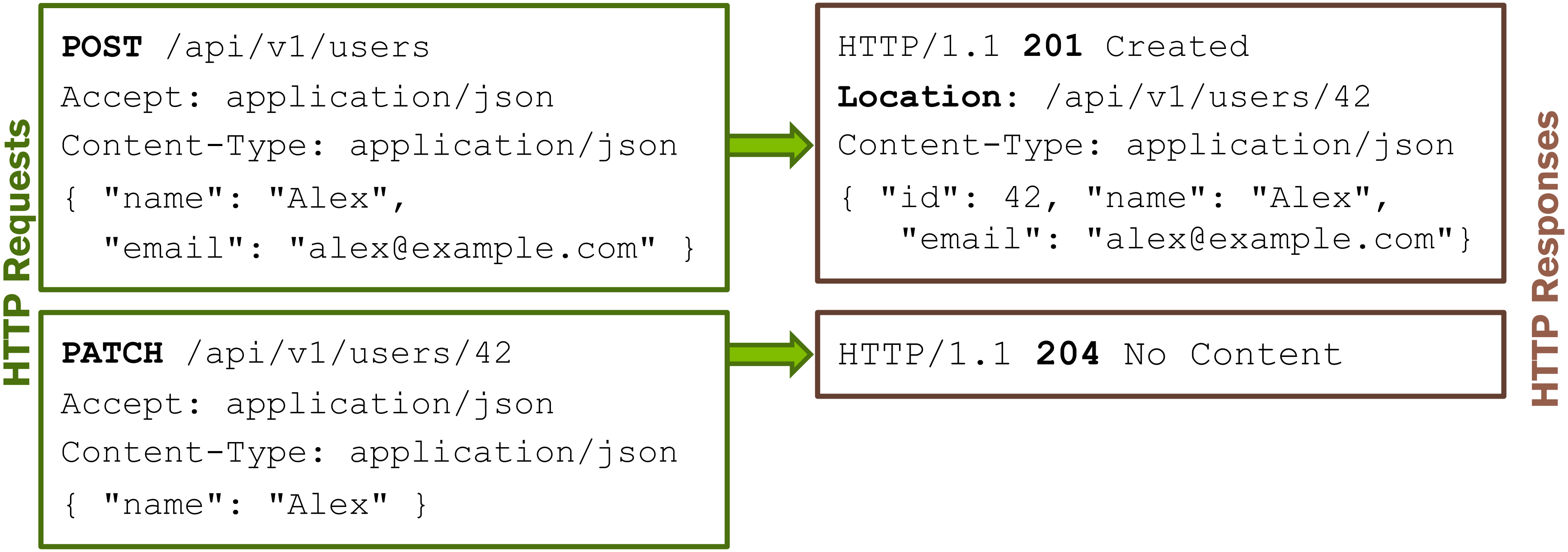
Correct Examples:

```
get  /users
get  /users/42
get  /users/42/orders/7
get  /users?page=3
get  /v1/users/42
```

```
                              payload
post    /users           {… user obj…}
put     /users/42        {… user obj…}}
patch   /users/42        { "blocked": true}
delete  /users
```

Uniform Interface: Manipulation Through Representations

- **Description:** Clients change state by sending resource representations
  - Client sends JSON; server persists state
- Modifying methods: **POST**=create; **PUT**= replace; **PATCH**=partial changes
- Responses: 201 + Location for creates; 200 with body or 204 for updates
- Validate input; on invalid data return 400 (malformed) or 422 (validation)



HTTP Requests

```
POST /api/v1/users
Accept: application/json
Content-Type: application/json
{ "name": "Alex",
  "email": "alex@example.com" }
```

```
PATCH /api/v1/users/42
Accept: application/json
Content-Type: application/json
{ "name": "Alex" }
```

HTTP Responses

```
HTTP/1.1 201 Created
Location: /api/v1/users/42
Content-Type: application/json
{ "id": 42, "name": "Alex",
   "email": "alex@example.com"}
```

```
HTTP/1.1 204 No Content
```

Uniform Interface: Self-Descriptive Messages

- **Description:** Messages carry intent, format, and outcome within themselves
- HTTP methods convey intent:

  - Safety & idempotency: **Safe** = no state change (GET).

  Idempotent = same result on repeat (PUT/DELETE; GET) POST non-idempotent; PATCH varies.

GET	Read
POST	Create
PUT	Replace
PATCH	Partial Changes
DELETE	Remove

- Content negotiation via Accept/Content-Type headers (typically application/json)
- Status codes communicate results; also, include helpful headers on the response

Common status codes:

200 OK successful read/update

201 Created new resource; include Location

204 No Content — success with no body (e.g., PATCH/DELETE)

401 Unauthorized

403 Forbidden authenticated but not allowed

404 Not Found resource doesn’t exist

422 Unprocessable Entity validation failed

429 Too Many Requests rate limit (Retry-After)

500 Internal Server Error server fault

Uniform Interface: HATEOAS

- HATEOAS - **H**ypermedia **a**s **t**he **E**ngine **o**f **A**pplication **S**tate
- **Description**: Hypermedia links guide clients to the next valid actions
- Use link relations (rel): self, collection, item, related, next/prev
- Clients follow links, not hard-coded paths; servers can evolve URLs safely
- Put links in representations and/or the Link header; on 201, use Location
- Optional in pragmatic REST — add links where they reduce coupling (pagination, actions).

Example of response data with HATEOAS:

```
{
  "id": 7,
  "links": [
    { "rel": "self", "href": "/orders/7" },
    { "rel": "items", "href": "/orders/7/items" },
    { "rel": "cancel", "href": "/orders/7/cancellations" }
  ]
}
```

API Authentication

- Token-based authentication is the most common auth API strategy
- Stateless requests; no server-side sessions
- Issue token at login; store securely on the client
- Send `Authorization: Bearer <token>` header over HTTPS on every request
- Server validate token; if no access: return 401 (unauthenticated), 403 (forbidden)
- Short lifetimes; support rotation and revocation
- API Authentication flow:

1. POST /login → 200 + token
2. Client stores token (securely)
3. HTTP request; add `Authorization: Bearer <token>` header
4. Server validates token + scopes → 200 or 401/403
5. POST /logout (revoke) → 204

REST API With Laravel

Laravel Practical Implementation Guidelines

REST API with Laravel

- Laravel supports server-rendered web apps and REST API backends.
- To prepare Laravel as a REST API server, run the following command:

```
php artisan install:api
```

  - Installs Sanctum for token auth
  - Adds `routes/api.php` file for API routes

Next slides outline API-specific guidelines (assumes Laravel basics)

REST API with Laravel - Guidelines

- Define API routes in `routes/api.php`
  - All routes are auto-prefixed with /api and use the api middleware group (stateless, rate limiting)
- Models (Eloquent) are the same as in web apps
- Controllers return JSON data instead of views/redirects
  - Prefer API Resources to shape responses
- Validation implementation is the same
  - Recommendation: use FormRequest classes
  - Validate request bodies for POST/PUT/PATCH
  - Behavior differs by Accept header:
    - text/html (web apps) - redirect back with session error messages (web flow)
    - application/json (API) - respond with 422 Unprocessable Entity and JSON error details
  - https://laravel.com/docs/validation

REST API with Laravel - Guidelines

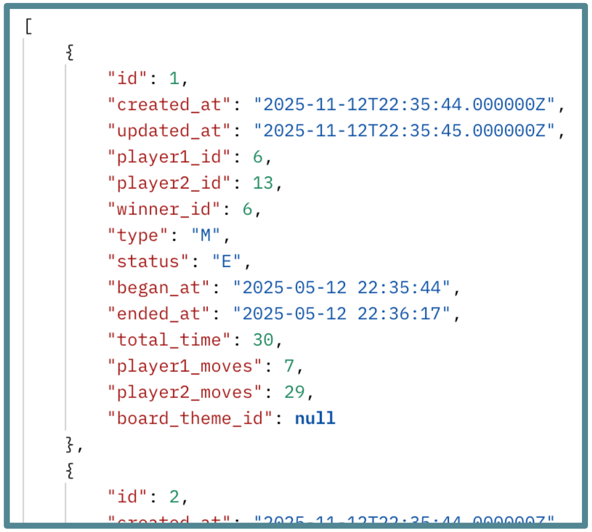
- Authentication: use Sanctum instead of session-based auth
  - Stateless tokens; no sessions
  - POST `/login` issues tokens; POST `/logout` revokes tokens
  - Routes protected by `auth:sanctum` middleware require a valid token
  - Send token on every request with header: `Authorization: Bearer <token>`
  - https://laravel.com/docs/sanctum

- Authorization implementation is the same — use gates & policies
  - Protect endpoints based on the user identity associated with the token
  - Combine `auth:sanctum` with `can:*` middleware or `$this->authorize()`
  - Access denials return 403 Forbidden (JSON in APIs) response
  - https://laravel.com/docs/authorization

REST API with Laravel - Guidelines

- Use API Resources to transform Eloquent models into stable JSON shapes
  - Hide internals; rename/compute/format fields; add links/meta; define custom structures
  - Return a single resource or a collection (pagination adds meta/links)
  - One Eloquent model can map to multiple resources (for different shapes/contexts)
  - Prefer resources at the API boundary (don’t return raw models)
  - https://laravel.com/docs/eloquent-resources

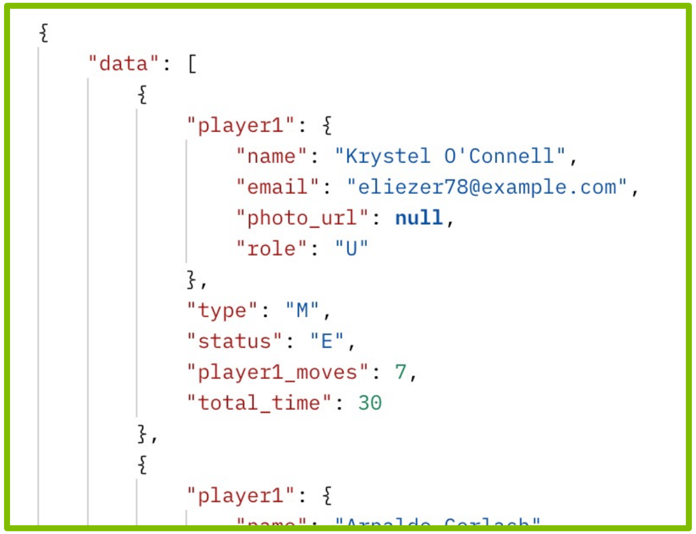
Raw data



```
[
    {
        "id": 1,
        "created_at": "2025-11-12T22:35:44.000000Z",
        "updated_at": "2025-11-12T22:35:45.000000Z",
        "player1_id": 6,
        "player2_id": 13,
        "winner_id": 6,
        "type": "M",
        "status": "E",
        "began_at": "2025-05-12 22:35:44",
        "ended_at": "2025-05-12 22:36:17",
        "total_time": 30,
        "player1_moves": 7,
        "player2_moves": 29,
        "board_theme_id": null
    },
    {
        "id": 2,
        "created_at": "2025-11-12T22:35:44.000000Z"
```

API Resource Transformation

Transformed data



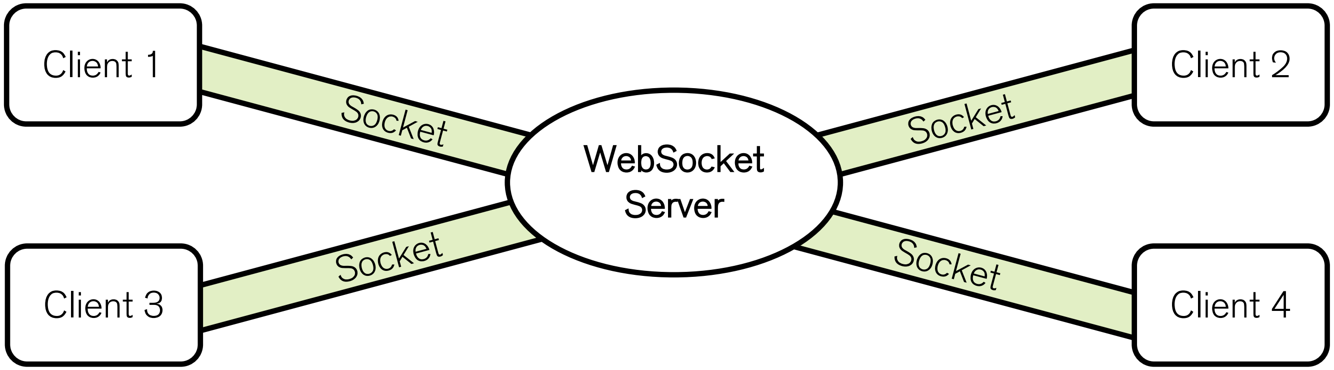
```
{
    "data": [
        {
            "player1": {
                "name": "Krystel O'Connell",
                "email": "eliezer78@example.com",
                "photo_url": null,
                "role": "U"
            },
            "type": "M",
            "status": "E",
            "player1_moves": 7,
            "total_time": 30
        },
        {
            "player1": {
                "name": "Arnaldo Gerlach"
```

WebSockets

Core Concepts

WebSockets

- WebSocket is a web standard (RFC 6455) supported by modern browsers
  - Browsers act as WebSocket clients via the W3C WebSocket API.
  - Native platforms (Android, iOS, Windows, etc.) also provide WebSocket clients via their SDKs/libraries.
- Bidirectional, full-duplex messaging over a persistent connection between a WebSocket server and a client.
  - Clients do not connect directly to each other — no peer-to-peer
- The WebSocket server handles many simultaneous sockets (multiple clients).
  - It coordinates communication between clients.



WebSocket Server Implementations & Services

- Node.js libraries
  - Socket.IO (events, rooms, auto-reconnect)
  - ws (minimal, fast)
  - uWebSockets.js (high-performance)
- Frameworks / platforms
  - Laravel Reverb (Laravel)
  - ASP.NET Core SignalR (.NET)
  - Spring WebSocket + STOMP (Java)
  - Phoenix Channels (Elixir)
  - Django Channels / FastAPI WebSockets (Python)
  - Gorilla/WebSocket ou nhooyr/websocket (Go)
- Managed services
  - Pusher Channels, Ably Realtime, PubNub
  - AWS API Gateway WebSocket, Azure Web PubSub, Cloudflare Durable Objects/WebSockets

WebSockets with Socket.IO

WebSockets Implementation Guidelines using Socket.IO

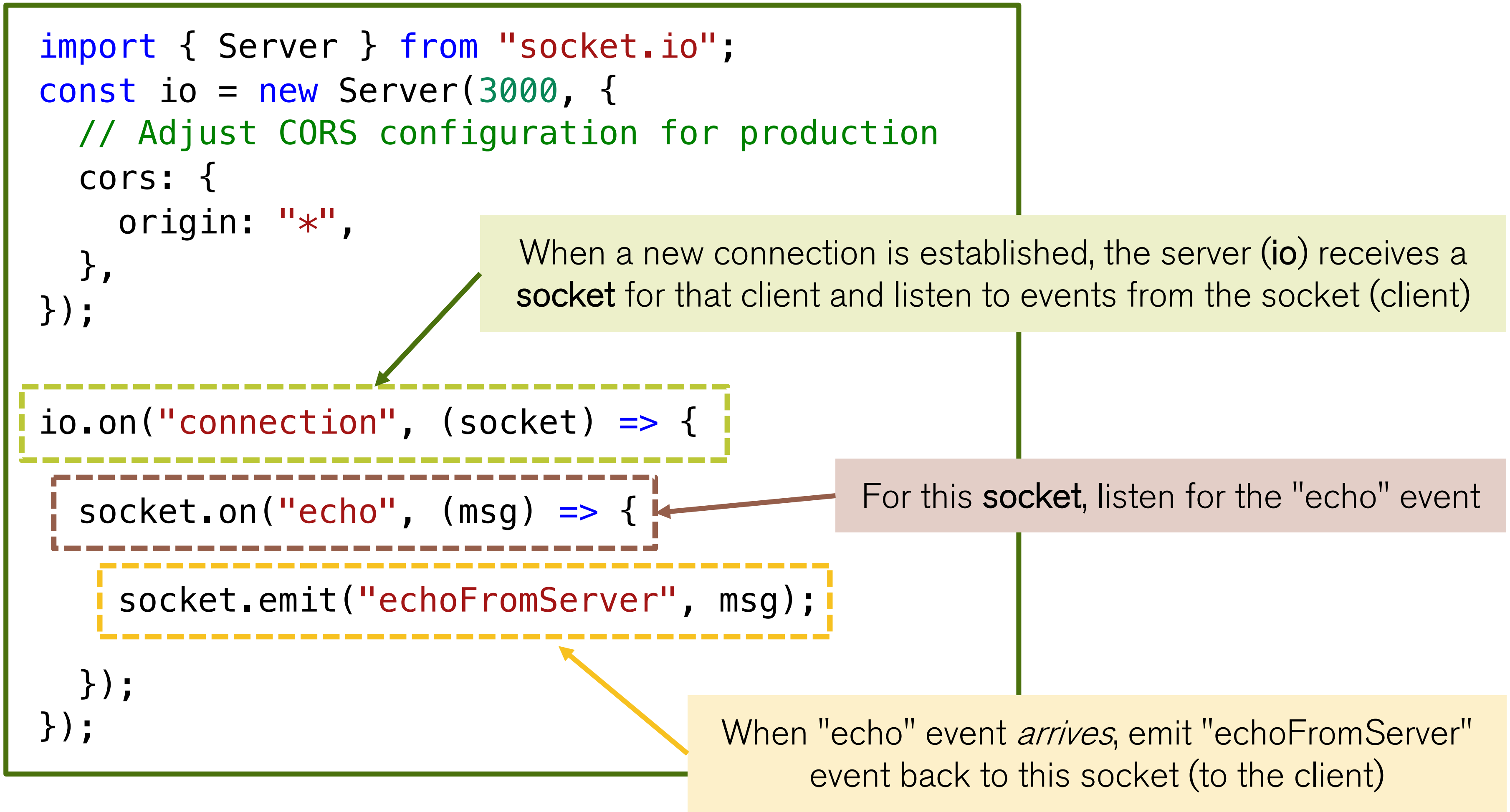
Socket.IO

- JavaScript library for real-time, event-based communication
- Paired libraries: client (browser) and server (Node.js) with a consistent API
- Uses WebSocket when available; falls back to HTTP long-polling
- Features: events, rooms, namespaces, auto-reconnect, acknowledgments, middleware
- Bidirectional messaging: client and server can both emit and listen to events
  - `emit()` - send a named event with data to the other side.
    - Use it to announce actions/updates (e.g., “join”, “update”, “chat”, "play").
  - `on()` - define how to handle a named event when it arrives.
    - Socket.IO runs your handler with the event’s data; works on client and server.

- https://socket.io/

Socket.IO Server

- Socket.IO server example (Node.js):



```
import { Server } from "socket.io";
const io = new Server(3000, {
  // Adjust CORS configuration for production
  cors: {
    origin: "*",
  },
});

io.on("connection", (socket) => {
  socket.on("echo", (msg) => {
    socket.emit("echoFromServer", msg);
  });
});
```

When a new connection is established, the server (**io**) receives a **socket** for that client and listen to events from the socket (client)

For this **socket**, listen for the "echo" event

When "echo" event *arrives*, emit "echoFromServer" event back to this socket (to the client)

Socket.IO

- To send a message from a client to the server, or from the server to a specific client (socket), **emit** a named event with `emit()` function

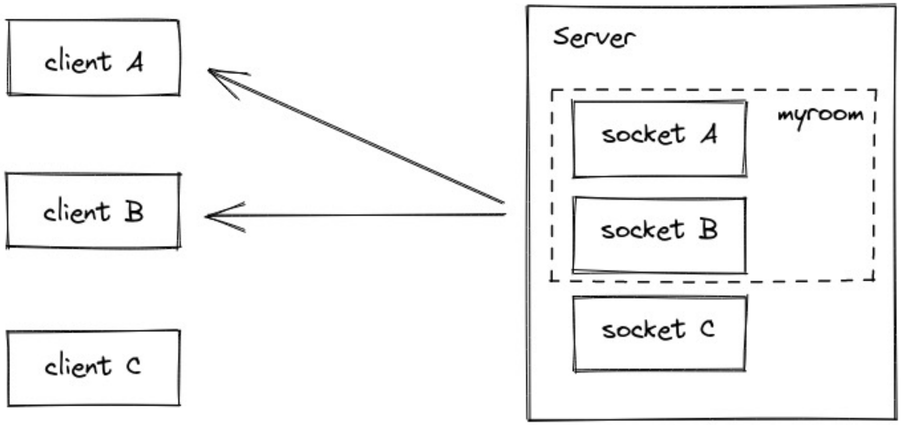
```
socket.emit("nameOfEvent", payload)
```

- To receive a message on either side (client or server), **listen** for the event with `on()` function - the handler (callback) fires when the message arrives

```
socket.on("nameOfEvent", (payload) => {
   // Event Handler
})
```

- **`socket`** refers to:
  - On the client: the connection to the server
  - On the server: the connection to a specific client (e.g., the sender)

Socket.IO - Rooms



- A room is a server-side channel that sockets can join and leave. It's used to broadcast to a subset of clients.
- Rooms are managed exclusively by the server
- A socket can join multiple rooms simultaneously
  - Join a room:

```
socket.join("game432")
```

  - Leave the room:

```
socket.leave("game432")
```

  - Emit a message to all clients (sockets) in the room:

```
io.to("game432").emit("nameOfEvent", payload)
```

Socket.IO Server API - Summary

- Assuming **io** is the server instance and **socket** is the "sender" client

sender only:

```
socket.emit('msg', data)
```

all clients:

```
io.emit('msg', data)
```

all except sender:

```
socket.broadcast.emit('msg', data)
```

all in room1:

```
io.to('room1').emit('msg', data)
```

all clients not in room1:

```
io.except('room1').emit('msg', data)
```

all in room1 except sender:

```
socket.to('room1').emit('msg', data)
```

all in room1 and/or room2, except sender:

```
socket.to(['room1', 'room2']).emit('msg', data)
```

all in room1 and/or room2, but not on room3:

```
io.to(['room1', 'room2']).except('room3').emit(...)
```

Socket.IO - Acknowledgements

- Per-event optional callback confirming receipt/result of an emitted event
- Works both ways: client ⇔ server (same API on both sides)
- Pass a function as the last argument to emit(); the receiver invokes it once

```
// Emitter
socket.emit('eventName', arg1, arg2, (ret1, ret2) => {
    // Acknowledgement callback
    //ret1, ret2 are returned by the receiver
});
```

```
// Receiver
socket.on('eventName', (arg1, arg2, ack) => {
    // Process and respond (call once)
    ack('ok', { id: 42 });
    // 'ok' and { id: 42 } go to the emitter's callback
});
```

Bibliography

- IETF, “RFC 6455: The WebSocket Protocol” https://datatracker.ietf.org/doc/html/rfc6455
- W3C, “WebSocket API” https://www.w3.org/TR/websockets/
- MDN Web Docs, “WebSocket (API)” https://developer.mozilla.org/en-US/docs/Web/API/WebSocket
- MDN Web Docs, “Server-Sent Events (EventSource)” https://developer.mozilla.org/en-US/docs/Web/API/EventSource
- JSON.org, “Introducing JSON” https://www.json.org
- JSON Schema, “Understanding JSON Schema” https://json-schema.org/understanding-json-schema
- OpenAPI Initiative, “OpenAPI Specification” https://spec.openapis.org/oas/latest.html
- Fielding, “Representational State Transfer (REST) — Dissertation Chapter” https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- MDN Web Docs, “HTTP request methods” https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
- MDN Web Docs, “HTTP response status codes” https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
- MDN Web Docs, “Content negotiation” https://developer.mozilla.org/en-US/docs/Web/HTTP/Content_negotiation
- GraphQL, “GraphQL — Learn” https://graphql.org/learn/
- GraphQL, “Schema & Type System” https://graphql.org/learn/schema/

Bibliography

- gRPC, “What is gRPC?” https://grpc.io/docs/what-is-grpc/
- gRPC, “gRPC-Web (Browser Clients)” https://grpc.io/docs/platforms/web/
- gRPC, “Status Codes” https://grpc.io/docs/guides/status-codes/
- Protocol Buffers, “Language Guide (proto3)” https://developers.google.com/protocol-buffers/docs/proto3
- Apache Avro, “Specification” https://avro.apache.org/docs/current/specification/
- Apache Kafka, “Introduction” https://kafka.apache.org/intro
- RabbitMQ, “Tutorials (Work Queues, Pub/Sub, Routing)” https://www.rabbitmq.com/tutorials/
- Socket.IO, “Documentation (v4)” https://socket.io/docs/v4/
- Laravel, “Sanctum” https://laravel.com/docs/sanctum
- Laravel, “Authorization (Gates & Policies)” https://laravel.com/docs/authorization
- Laravel, “Validation” https://laravel.com/docs/validation
- Laravel, “Eloquent: API Resources” https://laravel.com/docs/eloquent-resources
- Laravel, “Broadcasting” https://laravel.com/docs/broadcasting
- Laravel, “Reverb (WebSocket Server)” https://laravel.com/docs/reverb

Acknowledge

The authors acknowledge the use of ChatGPT (https://chatgpt.com/) and Google Gemini (https://gemini.google.com) as a tool supporting content development and text composition. Full responsibility for the final content rests with the authors.