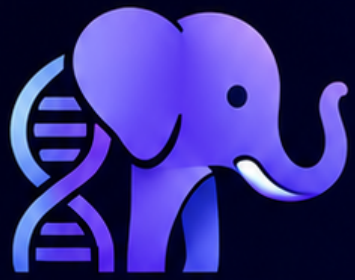

BioPHP

DOCUMENTATION

BioPHP Documentation

BioPHP is a PHP library dedicated to bioinformatics.

It provides simple and powerful tools to manipulate, analyze and visualize biological data.

SIMPLE
and intuitive

SPECIALIZED
for bioinformatics

POWERFUL
and high-performing

The BioAPI, BioPHP & BioTools ecosystem

A getting started guide for biologists

Amélie Duvernet

Version 1

Contents

Introduction

This guide is aimed at a biologist who already knows their scientific field but is discovering, for the first time, the software ecosystem made up of three complementary building blocks :

- **BioAPI** — a REST API (API Platform / Symfony) that exposes, for reading only, reference biological data : amino acids, nucleotides, codons, restriction enzymes, pKa scales, substitution matrices, etc. It is the single source of truth for the data.
- **BioPHP** (`amelaye/biophp`) — a PHP library / Symfony bundle that consumes this API, manipulates sequences (DNA, RNA, proteins) and can *parse* the main biological database formats (GenBank, Swiss-Prot, EMBL, PDB, PROSITE, ExPASy ENZYME, Entrez, TRANSFAC, KEGG…).
- **BioTools** (`amelaye/biotools`) — a Symfony bundle that dresses up biophp.org's historical "mini-tools" (DNA → protein translation, palindrome search, melting temperature, restriction enzyme digestion…) as ready-to-use forms and services. It relies entirely on BioPHP for data and calculations.

The dependency therefore runs in a single direction : **BioTools → BioPHP → BioAPI**. You can use BioAPI alone, BioPHP alone (with or without BioAPI depending on what you feed it), or all three together.

Each chapter that follows covers one of these three building blocks, with concrete examples geared towards "I'm looking for the function that does X".

Technical prerequisites

- A reasonably recent version of PHP (the exact constraint evolves with the project, so it is not pinned down here).
- Composer.
- For use as a Symfony bundle : nothing particular to set up in advance — BioTools installs by itself whatever it needs. The concrete details are covered in the chapters dedicated to each part.
- Outbound network access to `api.amelayes-biophp.net`, unless you host your own instance of BioAPI.

Installing in a Symfony application

This is the normal procedure if you are starting from an existing Symfony application (or a new project created with `symfony new`).

```
# The parsing and sequence library
composer require amelaye/biophp
```

```
# The ready-to-use mini-tools (optional)
composer require amelaye/biotools
```

These two commands also install everything BioPHP and BioTools need to work. We will see how to use them concretely, step by step, in the chapters dedicated to each.

Note

A complete Symfony demo application can be browsed online (https://demo.amelayes-biophp.net) if you want to see everything wired up in real Twig views.

"Vanilla" installation (without Symfony)

BioPHP is distributed as a Symfony bundle, but the classes that talk to the API and manipulate sequences have no hard dependency on the framework : they can be instantiated directly in a plain PHP script, provided you supply your own HTTP client (Guzzle) and serializer (JMS Serializer).

```
composer require amelaye/biophp guzzlehttp/guzzle jms/serializer
```

```php
<?php
require_once 'vendor/autoload.php';

use Amelaye\BioPHP\Domain\Sequence\Service\SequenceManager;

$client = new GuzzleHttp\Client([
    'base_uri' => 'https://api.amelayes-biophp.net',
]);
$serializer = JMS\Serializer\SerializerBuilder::create()->build();

$aminoApiManager  = new \Amelaye\BioPHP\Api\AminoApi($client, $serializer);
$nucleotidManager = new \Amelaye\BioPHP\Api\NucleotidApi($client, $serializer);
$elementManager   = new \Amelaye\BioPHP\Api\ElementApi($client, $serializer);

$sequenceManager = new SequenceManager(
    $aminoApiManager,
    $nucleotidManager,
    $elementManager
);

$aMirrors = $sequenceManager->findMirror(
    "AGGGAATTAAGTAAATGGTAGTGG",
    6,
    8,
    'E'
);

var_dump($aMirrors);
```

Warning

BioTools, on the other hand, needs at least a minimal Symfony setup to work as-is. For a fully standalone use, it is better to go through BioPHP alone. The chapter dedicated to it shows how to use BioTools regardless, even outside a real Symfony application.

Chapter 1

The API (BioAPI)

1.1 Overview

BioAPI is a REST API built with API Platform on top of Symfony. It exposes, in **read-only** mode, a set of reference tables in molecular biology : amino acids, nucleotides, codons, restriction enzymes, pKa scales, PAM250 substitution matrices, reduced protein alphabets, melting temperature parameters, and reagent vendors. It is the database that BioPHP, and indirectly BioTools, consume.

- Public base URL : `https://api.amelayes-biophp.net`
- Format : JSON-LD / Hydra (the API Platform standard) — each collection is returned under a `hydra:member` key.
- Every resource only exposes `GET` operations (collection and item) : this is a read-only API, not an editing one.
- Pagination is **disabled** : a request on a collection returns *all* records at once. Handy for loading a whole reference table into memory, but worth keeping in mind if you query the resource in a loop.

BioAPI also publishes an interactive Swagger interface, directly at its base URL, listing every resource and letting you try a request without writing a single line of code :

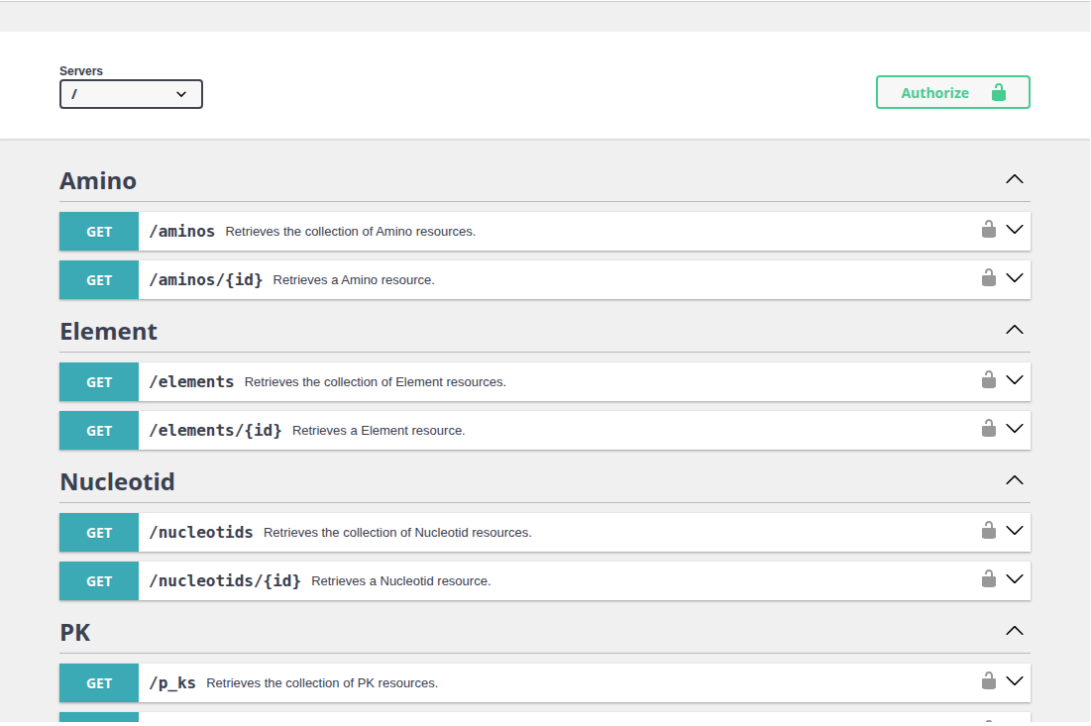

Figure 1.1: Swagger interface automatically generated by BioAPI, listing the available resources.

1.2 Authentication

BioAPI's Swagger documentation declares an API key, passed via the HTTP header :

```
X-AUTH-TOKEN: <your_token>
```

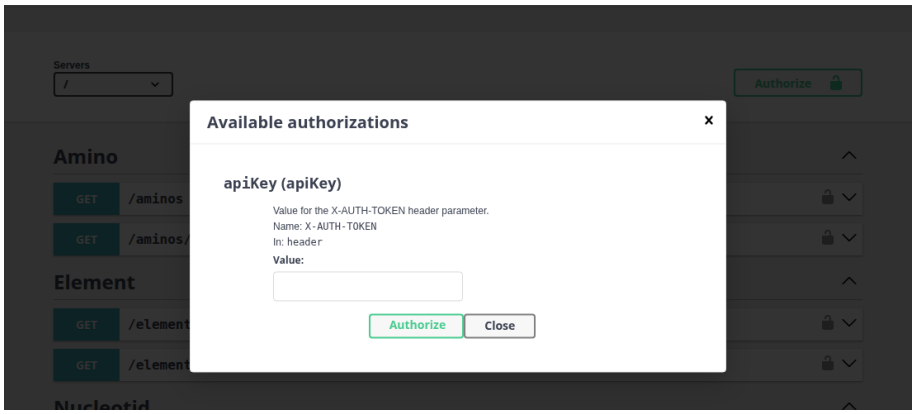

Figure 1.2: "Available authorizations" window of BioAPI's Swagger interface, with the exact name of the expected header.

BioPHP's PHP adapters (see chapter 2) accept this token as the optional third argument of their constructor.

Warning

Test performed from BioAPI's Swagger interface : a `GET /aminos` request sent **without** an `X-AUTH-TOKEN` header does return a `200` status with the full data. As things stand, authentication therefore does not appear to be enforced for reads on the public instance. Since this behaviour is not documented anywhere as intentional, it could change without notice : send the token as soon as you have one, just to be safe.

To complete

The procedure for obtaining a token for the public instance is not documented in the code — to be clarified in a future pass if it turns out to be needed for your use case, or to be ignored if you host your own instance without access control.

1.3 Available resources

The table below serves as a quick index : for each resource, the HTTP path, and the exposed fields. This is the entry point to check when you are looking for "where is such-and-such piece of data".

Resource	Path	Fields
Amino acids	`/aminos`	`id`, `name`, `name1Letter`, `name3Letters`, `weight1`, `weight2`, `residueMolWeight`
Chemical elements	`/elements`	`id`, `name` (e.g. *water*, *carbone*), `weight`
Nucleotides	`/nucleotids`	`id`, `letter` (A/T/G/C…), `complement`, `nature` (DNA/RNA), `weight`
pKa (EMBOSS)	`/p_ks/<id>` (single item)	`id` (scale name, e.g. *EMBOSS*), `nTerminus`, `cTerminus`, and the pKa values of side chains `k`, `r`, `h`, `d`, `e`, `c`, `y`
PAM250 matrix	`/pam250_matrix_digits`	`id` (index), `value` (substitution score)
Reduced alphabets	`/protein_reductions`	`id`, `alphabet` (e.g. *Murphy*), `letters`, `pattern`, `nature`, `reduction`, `description`
Tm — base stacking	`/tm_base_stackings`	`id`, `temperatureEnthalpy`, `temperatureEnthropy`
Codons (triplets)	`/triplets`	`id`, `triplet`
Codons per species	`/triplet_species`	`id`, `nature`, `triplets`, `tripletsGroups`
Type II enzymes	`/type_i_i_endonucleases`	`id` (enzyme name), `samePattern`, `recognitionPattern`, `computingPattern`, `lengthRecognitionPattern`, `cleavagePosUpper`, `cleavagePosLower`, `nbNonNBases`
Type IIb enzymes	`/type_i_ib_endonucleases`	same fields as type II

Resource	Path	Fields
Type IIs enzymes	`/type_i_is_endonucleases`	same fields as type II
Vendors	`/vendors`	`id`, `vendor`
Vendor links	`/vendor_links`	`id`, `name`, `link`

Note

The three restriction enzyme resources have paths that literally mirror the PHP class name (`TypeIIEndonuclease` → `/type_i_i_endonucleases`), due to API Platform's automatic pluralisation convention on a name containing consecutive capital letters. This is a detail not worth guessing : it is better to copy these paths as-is.

1.4 Example requests

1.4.1 From the command line

Fetching the full list of amino acids :

```
curl -H "Accept: application/ld+json" \
     https://api.amelayes-biophp.net/aminos
```

Fetching a specific pKa scale record :

```
curl -H "Accept: application/ld+json" \
     https://api.amelayes-biophp.net/p_ks/EMBOSS
```

A collection response looks like this (real excerpt, Hydra format — note the first entry `STOP`, which represents stop codons rather than an actual amino acid) :


```
{
  "@context": "/contexts/Amino",
  "@id": "/aminos",
  "@type": "hydra:Collection",
  "hydra:totalItems": 26,
  "hydra:member": [
    {
      "@id": "/aminos/*",
      "@type": "Amino",
      "id": "*",
      "name": "STOP",
      "name1Letter": "*",
      "name3Letters": "STP",
      "weight1": 0,
      "weight2": 0
    },
    {
      "@id": "/aminos/A",
      "@type": "Amino",
      "id": "A",
      "name": "Alanine",
      "name1Letter": "A",
      "name3Letters": "Ala",
```

```
      "weight1": 89.09,
      "weight2": 89.09,
      "residueMolWeight": 71.07
    }
  ]
}
```

The same round trip seen from BioAPI's Swagger interface ("Try it out" button, then "Execute") :

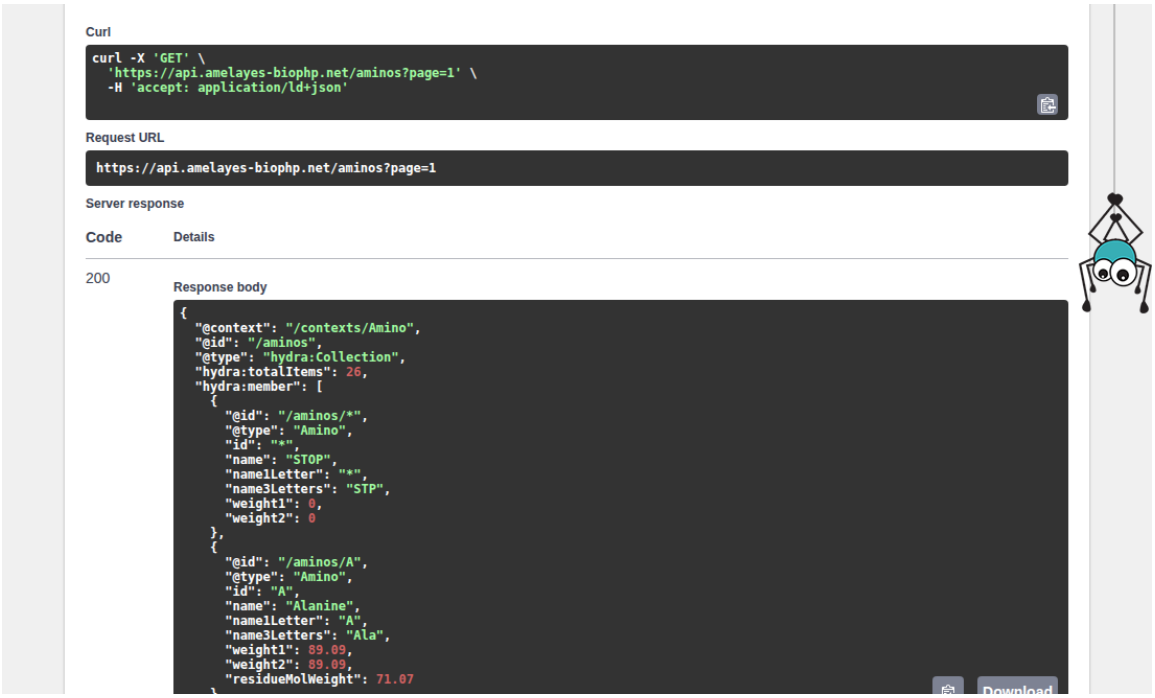

Figure 1.3: `GET /aminos` request executed from BioAPI's Swagger interface, with its actual response.

1.4.2 In PHP, with Guzzle

Without going through BioPHP's adapters — useful if you want to consume the API from a fully independent script :

```php
<?php
require_once 'vendor/autoload.php';

$client = new GuzzleHttp\Client([
    'base_uri' => 'https://api.amelayes-biophp.net',
]);

$response = $client->get('/nucleotids', [
    'headers' => ['Accept' => 'application/ld+json'],
]);

$data = json_decode($response->getBody()->getContents(), true);

foreach ($data['hydra:member'] as $nucleotid) {
    printf(
        "%s (%s) complement: %s, weight: %s\n",
        $nucleotid['letter'],
```

```
        $nucleotid['nature'],
        $nucleotid['complement'],
        $nucleotid['weight']
    );
}
```

1.4.3 A "biologist" example: locating a restriction enzyme's cut sites

A concrete "parsing" example on the API side : fetching a restriction enzyme's properties via `/type_i_i_endonucleases` (its recognition site, its cleavage positions) and using them to manually locate the cut sites in a sequence, without going through BioTools' ready-made digestion tool (see §3.4.15 for the "turnkey" version).

```php
<?php
require_once 'vendor/autoload.php';

$client = new GuzzleHttp\Client([
    'base_uri' => 'https://api.amelayes-biophp.net',
]);

// 1. Fetch type II enzymes to check a cleavage site
$response = $client->get('/type_i_i_endonucleases/EcoRI', [
    'headers' => ['Accept' => 'application/ld+json'],
]);
$ecoRI = json_decode($response->getBody()->getContents(), true);

printf(
    "EcoRI recognizes %s, cuts at position %d/%d\n",
    $ecoRI['recognitionPattern'],
    $ecoRI['cleavagePosUpper'],
    $ecoRI['cleavagePosLower']
);

// 2. Search for this site in a sequence
$sequence = "GGAATTCGGAATTCAAGCTT";
$site = $ecoRI['recognitionPattern']; // e.g. "GAATTC"
$positions = [];
$offset = 0;
while (($pos = strpos($sequence, $site, $offset)) !== false) {
    $positions[] = $pos;
    $offset = $pos + 1;
}

printf("Sites %s found at positions: %s\n", $site, implode(', ',
    $positions));
```

Note

The exact identifier to use in the URL (`EcoRI` above) corresponds to the entity's `id` field — check via `GET /type_i_i_endonucleases` if the exact name of the enzyme you are interested in is not known in advance.

Chapter 2

BioPHP

2.1 Overview

BioPHP breaks down into four families of tools, covered in the sections that follow:

- **Reading biological database files** — one reader per format (GenBank, Swiss-Prot, PROSITE, TRANSFAC, KEGG…), 31 in total.
- **Manipulating sequences** — translation, complementation, motif search, alignment, restriction enzymes…
- **A small toolbox** — a handful of biology and statistics functions that come up often.
- **Talking to BioAPI** — the classes that fetch the reference data presented in Chapter 1 (amino acids, nucleotides, enzymes…), used behind the scenes by the three families above.

The first family — reading files — takes up most of this chapter: it has the largest number of classes (31 readers), and it's generally the first stop for a biologist who has a downloaded file on disk and wants to extract information from it.

Note

This whole family rests on two folders that mirror each other: `Domain/Database` carries the generic mechanism (the factory that picks the right reader, the collection manager for large files), and `Domain/Parser` carries the 31 readers themselves, plus the few objects that some of them return in addition to the sequence. Both folders share the same internal organization: a `Service` subfolder for the classes that do the work, `Entity` for the objects they handle or return, and `Interfaces` for each one's contract. You don't need to remember this organization to use BioPHP — a single entry point is enough, see the next section — but it explains the full class paths cited in the code examples.

2.2 Installation and configuration

2.2.1 Installation

```
composer require amelaye/biophp
```

In Symfony, register the bundle in `config/bundles.php`:

```
1 return [
2     // ...
```

```
    JMS\SerializerBundle\JMSSerializerBundle::class => ['all' => true],
    Amelaye\BioPHP\AmelayeBioPHPBundle::class => ['all' => true],
];
```

Note

The bundle itself activates the Doctrine ORM and registers its own entities (`Domain/Database/Entity` and `Domain/Sequence/Entity`) with it, via a dependency-injection `prepend()`: you therefore have *nothing* to declare on the `doctrine.yaml` side for this mapping to be picked up, as long as `doctrine/orm` and `doctrine/doctrine-bundle` are installed — they already are, as dependencies of BioPHP.

In a "vanilla" installation (without Symfony), you instantiate the classes under `Domain/` that you need directly. For simple file reading (§2.3.1), no database is needed; to index large files or to save what a reader has extracted, see the next section.

2.2.2 Persistence: installing the tables in an existing MySQL database

BioPHP only needs a database for two purposes: indexing a large file so you can retrieve a specific record without rereading the whole thing (`DatabaseManager`, §2.3.1), and saving what a reader has extracted so you can read it back later without going through the original file again (`RecordManager`, §2.3.2). If your files fit in memory and you reread them every time, you can skip this section entirely: nothing in this chapter requires using a database.

When you do need it, BioPHP maps twelve Doctrine entities, which translate into as many tables:

Table	Role
`collection`	A named set of files indexed together (an imported batch).
`collection_element`	The index of a record within a file in the collection: its identifier, the file, and the line where it starts. Used by `DatabaseManager::fetch()`.
`parsed_record`	What a reader extracted from a record, stored as JSON. Used by `RecordManager` (§2.3.2).
`sequence`, `feature`, `reference`, `author`, `accession`, `keyword`, `gb_sequence`, `sp_databank`, `src_form`	The nine common objects from §2.3.3, mapped as Doctrine entities so the schema is complete and tested — **but no BioPHP code writes to them**, see the warning box below.

Warning

The tables `sequence`, `feature`, `reference`, etc. exist in the schema but stay empty unless you populate them yourself: the 31 readers in this chapter return these objects in memory, for you to read via their getters, but never persist them on their own. The only automatic persistence BioPHP offers goes through `RecordManager` and the `parsed_record` table (§2.3.2). Don't be surprised, then, to find these nine tables empty after reading dozens of files: that's expected behavior.

In Symfony

Set the `DATABASE_URL` environment variable to point to your existing MySQL database:

```
# .env.local
DATABASE_URL="mysql://user:password@127.0.0.1:3306/my_database?
    serverVersion=8.0"
```

Then create the missing tables:

```
# fresh database, or an existing database with none of the tables above
$ bin/console doctrine:schema:create

# existing database that already has other tables (yours, or a
# previous BioPHP install to upgrade):
# only creates/modifies what's missing, without touching the rest
$ bin/console doctrine:schema:update --force
```

Note

`doctrine:schema:update -force` compares the entity mapping against the schema actually present in the database and applies only the difference: this is the command to prefer whenever the database already contains tables that aren't yours (your application's, or another bundle's). It never drops a table that's no longer mapped, unless you explicitly add `-complete`.

In vanilla

Without Symfony, you build the `EntityManager` yourself, pointing the attribute mapping at BioPHP's two entity folders, then ask Doctrine to create or update the schema:

```
<?php
require_once 'vendor/autoload.php';

use Doctrine\DBAL\DriverManager;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\Mapping\UnderscoreNamingStrategy;
use Doctrine\ORM\ORMSetup;
use Doctrine\ORM\Tools\SchemaTool;

$config = ORMSetup::createAttributeMetadataConfiguration(
    [
        __DIR__ . '/vendor/amelaye/biophp/Domain/Database/Entity',
        __DIR__ . '/vendor/amelaye/biophp/Domain/Sequence/Entity',
    ],
    true // development mode; set to false in production
);
$config->setNamingStrategy(new UnderscoreNamingStrategy(CASE_LOWER, true
    ));

$connection = DriverManager::getConnection([
    'driver'   => 'pdo_mysql',
    'host'     => '127.0.0.1',
    'port'     => 3306,
    'dbname'   => 'my_database',
    'user'     => 'user',
```

```
    'password' => 'password',
], $config);

$entityManager = new EntityManager($connection, $config);

// run once to create the missing tables in the existing database
// ('my_database' above), without touching its current tables
(new SchemaTool($entityManager))->updateSchema(
    $entityManager->getMetadataFactory()->getAllMetadata()
);
```

Warning

BioPHP does not ship a migrations library (`doctrine/migrations` is not a dependency): it's `doctrine:schema:update` (Symfony) or `SchemaTool::updateSchema()` (vanilla) that serves as the installation and schema-upgrade tool. If your project already uses `doctrine/migrations`, nothing stops you from folding BioPHP's entities into it alongside your own: they're mapped with attributes the same way.

Warning

If you're upgrading a BioPHP installation from before September 20, 2026: the `organism` field of `Sequence` moved from PHP serialized storage (an `array` type) to JSON, and several foreign keys and index names changed (the constraints named `prim_acc` on six different tables were colliding on SQLite and PostgreSQL, where index names are global). These tables must be recreated: `doctrine:schema:update -force` takes care of it, but back up any data you want to keep first.

2.3 Reading a database file

2.3.1 The reading mechanism, once and for all

Whatever the format, reading always works the same way: load the file line by line, hand those lines to `DatabaseReaderFactory` specifying the format, and get back an object — the *reader* — whose getters give access to the information read.

```
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('my_file.gb'); // an array, one line per entry
$reader = DatabaseReaderFactory::readDatabase('GENBANK', $lines);
```

This is the only call needed in the vast majority of cases. The format name (`'GENBANK'`, `'SWISSPROT'`, `'PROSITE'`...) is always the one that appears in the "Format" column of the index in section 2.5.

Behind the scenes, `DatabaseReaderFactory` delegates all the work to `DatabaseParserFactory`: it's this second class that knows the list of 31 supported formats (`getFormats()`), that maps a format name to the class that reads it (`getParserClass()`), and that builds a fresh instance of that class (`createParser()`). It's also the entry point to modify for anyone wanting to add a 32nd format: a class in `Domain/Parser/Service`, and one registration line in this factory.

Note

The readers themselves are not Symfony services — there's no declaration to make for them in the dependency injection container. That's what lets the call above work identically in Symfony and in a "vanilla" installation: in both cases, you simply call `DatabaseReaderFactory::readDatabase()`.

What all the readers have in common, at bottom, is knowing how to answer four questions about their format: where a record starts, where it ends, what its identifier is, and how to read the lines that make it up. It's this shared contract (the `ParseDatabaseInterface` interface) that lets `DatabaseManager` (below) split any file apart without knowing anything about the format it contains.

Note

A database file usually contains several records in a row, and the example above assumes `my_file.gb` contains only one. For a file containing thousands, `DatabaseManager` takes over: its `recording()` method scans the file once to locate where each record starts and ends, and indexes that information (via Doctrine, in the `Collection` and `CollectionElement` tables). Its `fetch($id)` method then retrieves a specific record by its identifier, without having to reread the whole file from the start, and returns you the same kind of reader as `DatabaseReaderFactory::readDatabase()`. This is the only part of BioPHP that requires a database: if your files fit in memory and you already know which file contains the record you're interested in, you don't need it.

2.3.2 Saving and rereading records

The mechanism described above (`recording()` / `fetch()`) indexes a file to locate a record in it, but goes back to that file to reread and reparse it on every call. BioPHP can also keep what a reader extracted directly, once and for all, thanks to `Amelaye\BioPHP\Domain\Database\Service\RecordManager` (see its installation in §2.2.2).

This mechanism is entirely generic: it works identically for the 31 formats in this chapter, including a future 32nd format you might add yourself, with no extra code — it simply reads the public getters of whatever the reader returned. Three operations, grouped behind the `RecordStorageInterface` interface:

Method	Role
`store($format, $lines)`	Parses a single record (the same array of lines as `readDatabase()`) and saves it to the database. A record already stored under the same identifier is updated rather than duplicated.
`storeFile($name, $format, ...$files)`	Parses every record in one or more files and saves them, grouped under a collection named $name (created if it doesn't already exist). Returns the number of records processed.
`find($format, $id)`	Retrieves an already-stored record, without going back through the original file. Returns `null` if it doesn't exist.

What gets stored is a `ParsedRecord` object, whose `getData()` returns an associative array: the same information as the reader's getters, but flattened and ready for JSON (`getGeneNames()`

becomes the `geneNames` key, `isTruePositive()` becomes `truePositive`, and an object encountered along the way — a reference, an atom… — is unpacked the same way, recursively).

```php
<?php
use Amelaye\BioPHP\Domain\Database\Service\RecordManager;

// $recordManager is autowired in Symfony (RecordStorageInterface);
// in vanilla: new RecordManager($entityManager, './data/');

// Store an entire Swiss-Prot file at once
$stored = $recordManager->storeFile('uniprot_sample', 'SWISSPROT', '
    sprot.dat');
echo "$stored records stored.\n";

// Retrieve a specific record without rereading the file
$record = $recordManager->find('SWISSPROT', 'TNFA_HUMAN');
if ($record !== null) {
    $data = $record->getData();
    echo $data['sequence']['description'] ?? '';
    foreach ($data['geneNames'] ?? [] as $synonyms) {
        echo implode(' / ', $synonyms) . "\n";
    }
}
```

Each card below recalls, in its "Persistence" paragraph, the format name to pass to `store()` / `storeFile()` / `find()` — it's the same name given to `readDatabase()`, in the same column of the index in section 2.5.

Warning

`storeFile()` parses and writes each record individually: for a very large file (tens of thousands of records), prefer running it from the command line or as a background job rather than within an HTTP request, even though a batched flush every 100 records keeps memory usage down.

2.3.3 Objects common to every reading

Most of the 31 readers — all those that describe an annotated sequence (GenBank, Swiss-Prot, EMBL foremost) — inherit from the same base class, `ParseDbAbstractManager`, and therefore return their information through the same set of objects, all under `Amelaye\BioPHP\Domain\Sequence\Entity`. This table serves as a shared reference for all the cards that follow: each one simply indicates which of these objects it actually populates.

Object	Returned by	Main fields
`Sequence`	`getSequence()`	`getPrimAcc()` (main accession), `getEntryName()`, `getSeqLength()`, `getMolType()`, `getDate()`, `getSource()`, `getOrganism()` (array, from taxonomic lineage to kingdom), `getDescription()`, `getSequence()` (the raw sequence), `getFragment()`

Object	Returned by	Main fields
`Feature`	`getFeatures()` (array)	`getFtKey()` (e.g. `"CDS"`, `"gene"`), `getFtFrom()`, `getFtTo()`, `getStrand()` (`"+"` or `"-"`), `getFtQual()` / `getFtValue()` (qualifier name and value), `getFtDesc()`
`Reference`	`getReferences()` (array)	`getRefno()`, `getTitle()`, `getJournal()`, `getMedline()`, `getPubmed()`, `getBaseRange()`, `getRemark()`, `getComments()`
`Author`	`getAuthors()` (array)	`getRefno()` (the reference it's attached to), `getAuthor()`
`Accession`	`getAccession()` (array)	`getAccession()` — the *secondary* accessions of a record (the primary one is already in `Sequence::getPrimAcc()`)
`Keyword`	`getKeywords()` (array)	`getKeywords()` — one keyword per object
`GbSequence`	`getGbSequence()`	`getVersion()`, `getNcbiGiId()`, `getStrands()`, `getTopology()`, `getDivision()`, `getSegmentNo()` / `getSegmentCount()` — not always populated, see each card
`SrcForm`	`getSrcForm()`	`getEntry()` — rarely used, kept for compatibility
`SpDatabank`	`getSpDatabank()` (array)	`getDbName()`, `getPid1()`, `getPid2()` — cross-references to other databases (Swiss-Prot's `DR` field)

2.4 Objects returned by certain parsers

Five formats — PDB, PROSITE, ExPASy ENZYME, ENTREZ and GENOME — describe information that the objects above don't provide for (atomic coordinates, a list of PDB references, a disease linked to an enzyme deficiency...), across seven objects in total (PDB alone accounts for three). Each therefore has its own object, in `Domain/Parser/Entity`:

Object	Returned by	Fields
`PdbAtom`	`PDB` (`getAtoms()`, `getHetAtoms()`)	`getSerial()`, `getName()`, `getAltLoc()`, `getResName()`, `getChainId()`, `getResSeq()`, `getX()` / `getY()` / `getZ()`, `getOccupancy()`, `getTempFactor()`, `getElement()`
`PdbHelix`	`PDB` (`getHelices()`)	`getHelixId()`, `getInitResName()`, `getInitChainId()`, `getInitSeqNum()`, `getEndResName()`, `getEndChainId()`, `getEndSeqNum()`, `getHelixClass()`, `getLength()`
`PdbSheet`	`PDB` (`getSheets()`)	`getSheetId()`, `getStrand()`, `getInitResName()`, `getInitChainId()`, `getInitSeqNum()`, `getEndResName()`, `getEndChainId()`, `getEndSeqNum()`
`PrositeDbRef`	`PROSITE` (`getDbRefs()`)	`getAccession()`, `getEntryName()`, `isTruePositive()`

Object	Returned by	Fields
ExpasyDisease	EXPASY_ENZYME (getDiseases())	getDisease(), getReference()
EntrezReference	ENTREZ (getReferences())	getRefNo(), getBaseRange(), getAuthors(), getTitle(), getJournal(), getMedline(), getPubmed(), getRemark()
GenomeReference	GENOME (getReferences())	getType(), getAuthors(), getTitle(), getJournal(), getVolume(), getPages(), getYear()

Note

Each of these objects has its own interface (PdbAtomInterface, PrositeDbRefInterface...) in Domain/Parser/Interfaces. You don't need them to read a file, but they let you, if you have a use for it, substitute your own implementation for the object provided by BioPHP without touching the reader itself.

2.5 The parsers, one by one

Each card below can be read independently of the others: jump directly to the one for the format you're interested in. The index below serves as a quick lookup table between a format name and what it describes, to help you find the right one if your file doesn't match any of the examples already covered.

Table 2.5: Index of the 31 supported formats

Format	Describes
SWISSPROT	Annotated protein sequence (UniProt/Swiss-Prot)
GENBANK	Annotated sequence (NCBI)
EMBL	Annotated sequence (EMBL)
PDB	3D structure (atomic coordinates, helices, sheets)
PROSITE	Motif / signature of a protein family
EXPASY_ENZYME	Enzyme by EC number
PDBSTR	Structural family derived from PDB
UNIGENE	Cluster of sequences expressing the same gene
PRINTS	Fingerprint (multiple motifs) of protein families
BLOCKS	Conserved motifs of protein families
AAINDEX	Physicochemical indices of amino acids
PRODOM	Families of protein domains
NCBI_LIT	Scientific journal reference set (NCBI)
PMD	Protein mutations (Protein Mutant Database)
HGBASE	Human biallelic genetic variants
PRF	Protein sequences (Protein Research Foundation)
PIR	Annotated protein sequences (Protein Information Resource)
EPD	Eukaryotic promoters (Eukaryotic Promoter Database)
GENOME	Genome sequencing statistics
ENTREZ	Genomic record (NCBI Entrez)
TRANSFAC_MATRIX	Position weight matrix (transcription factors)

Table 2.5 – continued

Format	Describes
TRANSFAC_GENE	Regulated gene
TRANSFAC_CLASS	Transcription factor class
TRANSFAC_CELL	Cell type
TRANSFAC_FACTOR	Transcription factor
TRANSFAC_SITE	Factor binding site
KEGG_COMPOUND	Chemical compound (metabolism)
KEGG_REACTION	Chemical reaction
KEGG_ENZYME	Enzyme (metabolism view)
KEGG_ORTHOLOG	Group of orthologous genes
KEGG_GENOME	Genome (metabolism view)

2.5.1 Swiss-Prot

A Swiss-Prot record (today UniProtKB/Swiss-Prot) describes a protein: its sequence, its accessions, its manually annotated structural and functional characteristics, its bibliographic references, and its cross-references to other databases. It's the best-annotated of the three major sequence formats (along with GenBank and EMBL) — the one to turn to first when annotation quality matters more than exhaustiveness.

Format to pass to `readDatabase()`: `'SWISSPROT'`

A record opens with an `ID` line and closes with a `//` line:

```
ID   TNFA_HUMAN     STANDARD;      PRT;   233 AA.
AC   P01375;
DT   01-NOV-1990 (REL. 16, CREATED)
DE   TUMOR NECROSIS FACTOR PRECURSOR (TNF-ALPHA).
GN   TNF.
OS   HOMO SAPIENS (HUMAN).
OC   EUKARYOTA; METAZOA; CHORDATA; MAMMALIA; PRIMATES.
RN   [1]
RP   X-RAY CRYSTALLOGRAPHY.
RA   ECK M.J., SPRANG S.R.;
RL   J. BIOL. CHEM. 264:17595-17605(1989).
DR   PDB; 1TNF; X-RAY.
KW   CYTOKINE; GLYCOPROTEIN.
FT   DOMAIN       1     35       CYTOPLASMIC.
SQ   SEQUENCE   233 AA;  25644 MW;
     MSTESMIRDV ELAEEALPKK TGGPQGSRRC LFLSLFSFLI VAGATTLFCL LHFGVIGPQR
//
```

What the reader returns: the common objects from section 2.3.3 (`getSequence()`, `getFeatures()`, `getReferences()`, `getAuthors()`, `getAccession()`, `getKeywords()`), plus:

Method	Content
`getSpDatabank()`	Array of `SpDatabank` — cross-references to other databases (`DR` field)
`getGeneNames()`	Array of gene names, grouped by synonyms (`GN` field)
`getSequpdDate()`	Date of the last sequence update (`DT` field)
`getNotupdDate()`	Date of the last annotation update (`DT` field)

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('tnfa_human.txt');
$reader = DatabaseReaderFactory::readDatabase('SWISSPROT', $lines);

$sequence = $reader->getSequence();
echo $sequence->getEntryName();      // "TNFA"
echo $sequence->getDescription();    // "TUMOR NECROSIS FACTOR PRECURSOR
    ..."
echo $sequence->getSequence();       // the protein sequence itself

// Gene names, grouped by synonyms
foreach ($reader->getGeneNames() as $synonyms) {
    echo implode(' / ', $synonyms) . "\n";
}

// Annotated sites and domains (FT)
foreach ($reader->getFeatures() as $feature) {
    printf(
        "%s: from %d to %d\n",
        $feature->getFtKey(),   // e.g. "DOMAIN", "ACT_SITE", "BINDING"
        $feature->getFtFrom(),
        $feature->getFtTo()
    );
}

// Cross-references to other databases (DR), for example to find
// the PDB structures associated with this protein
foreach ($reader->getSpDatabank() as $ref) {
    if ($ref->getDbName() === 'PDB') {
        echo "PDB structure: " . $ref->getPid1() . "\n";
    }
}

echo "Last sequence update: " . $reader->getSequpdDate();
```

What this is actually useful for: getting the protein sequence and its description in one line; listing a gene's synonyms; walking through manually annotated functional sites (domains, active sites, binding sites) to place them on the sequence; quickly finding, via `getSpDatabank()`, the PDB, InterPro or Pfam identifiers linked to a protein without having to query those databases separately; knowing how fresh the annotation is before trusting it.

Persistence: `$recordManager->find('SWISSPROT', $id)` retrieves an already-stored record; `storeFile($name, 'SWISSPROT', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.2 GenBank

A GenBank record also describes an annotated sequence, but with a different emphasis than Swiss-Prot: where Swiss-Prot documents a protein in detail, GenBank documents the genomic context of a nucleic sequence — where the genes, exons, and CDSs sit on the sequenced strand. It's the NCBI's historical format, and the most common one for DNA and RNA.

Format to pass to `readDatabase()`: `'GENBANK'`

A record opens with a `LOCUS` line and closes with a `//` line:

```
LOCUS       AB012345                233 bp    mRNA    linear   PRI 01-
   JAN-2020
DEFINITION  Homo sapiens mRNA for tumor necrosis factor, complete cds.
ACCESSION   AB012345
VERSION     AB012345.1  GI:123456789
KEYWORDS    TNF.
SOURCE      Homo sapiens (human)
  ORGANISM  Homo sapiens
            Eukaryota; Metazoa; Chordata; Mammalia; Primates.
REFERENCE   1  (bases 1 to 233)
  AUTHORS   Eck M.J., Sprang S.R.
  TITLE     The structure of TNF
  JOURNAL   J. Biol. Chem. 264:17595-17605(1989)
FEATURES             Location/Qualifiers
     source          1..233
                     /organism="Homo sapiens"
     gene            1..233
                     /gene="TNF"
     CDS             1..233
                     /product="tumor necrosis factor"
ORIGIN
        1 atgagcacag aaagcatgat ccgggacgtg gagctggccg
//
```

What the reader returns: the common objects from section 2.3.3, plus `getGbSequence()` — a `GbSequence` with, for GenBank, the full version (`getVersion()`), the `GI` identifier (`getNcbiGiId()`), the strand type, topology and division.

Warning

The `FEATURES` table can describe dozens of different block types in a real GenBank file. The BioPHP reader explicitly recognizes only five: `source`, `gene`, `exon`, `CDS` and `misc_feature`. Other types (`mRNA`, `intron`, `polyA_site`…) are walked through without error but don't show up in `getFeatures()`.

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('sequence.gb');
$reader = DatabaseReaderFactory::readDatabase('GENBANK', $lines);

$sequence = $reader->getSequence();
echo $sequence->getPrimAcc();        // the accession number, e.g. "
    AB012345"
echo $sequence->getDescription();    // one-line description (DEFINITION)
echo $sequence->getOrganism();       // the organism's classification
echo $sequence->getSequence();       // the sequence itself (A, T, G, C
    ...)

$gb = $reader->getGbSequence();
echo $gb->getVersion();              // "AB012345.1"
echo $gb->getNcbiGiId();             // "123456789"

foreach ($reader->getFeatures() as $feature) {
    printf(
        "%s: from %d to %d (%s)\n",
```

```
        $feature->getFtKey(),    // "source", "gene", "exon" or "misc_feature"
        $feature->getFtFrom(),
        $feature->getFtTo(),
        $feature->getStrand()    // "+" or "-"
    );
}

foreach ($reader->getReferences() as $reference) {
    echo $reference->getTitle() . " (" . $reference->getJournal() . ")\n";
}
```

What this is actually useful for: extracting the sequence and pinpointing the organism it comes from; listing genes, CDSs and exons with their positions and strand, for example to cut out the coding region and translate it (see `translate()`, next section); comparing the version identifier (`VERSION`/`GI`) between two downloads of the same record to detect an update; retrieving the bibliography attached to a sequence.

Persistence: `$recordManager->find('GENBANK', $id)` retrieves an already-stored record; `storeFile($name, 'GENBANK', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.3 EMBL

The EMBL format (the European database, today ENA) describes the same kind of information as GenBank — annotated sequence, organism, feature table, bibliography — but with Swiss-Prot's two-letter tag grammar (`ID`, `AC`, `DE`…) rather than GenBank's full keywords. Useful for files coming from ENA rather than NCBI, for the same purposes as GenBank.

Format to pass to `readDatabase()`: `'EMBL'`

A record opens with an `ID` line and closes with a `//` line:

```
ID   AB012345; SV 1; linear; mRNA; STD; HUM; 233 BP.
AC   AB012345;
DT   01-JAN-2020 (Rel. 1, Created)
DE   Homo sapiens mRNA for tumor necrosis factor, complete cds.
KW   TNF.
OS   Homo sapiens (human)
OC   Eukaryota; Metazoa; Chordata; Mammalia; Primates.
RN   [1]
RP   1-233
RX   PUBMED; 2547453.
RA   Eck M.J., Sprang S.R.;
RT   "The structure of TNF";
RL   J. Biol. Chem. 264:17595-17605(1989).
FT   source          1..233
FT                   /organism="Homo sapiens"
FT   CDS             1..233
FT                   /product="tumor necrosis factor"
SQ   Sequence 233 BP;
     atgagcacag aaagcatgat ccgggacgtg gagctggccg
        40
//
```

What the reader returns: the same common objects as Swiss-Prot and GenBank (section 2.3.3). The EMBL reader also fills in `getGbSequence()` — topology, division and version are populated there, but not the `GI` identifier (specific to GenBank) nor the strand count.

Note

Unlike GenBank, the EMBL reader imposes no closed list of `FEATURES` types: every `FT` block encountered is read and shows up in `getFeatures()`, whatever its type. This is the opposite of the limitation noted in the warning box on the GenBank card.

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('sequence.embl');
$reader = DatabaseReaderFactory::readDatabase('EMBL', $lines);

$sequence = $reader->getSequence();
echo $sequence->getPrimAcc();        // "AB012345"
echo $sequence->getDescription();
echo $sequence->getSequence();

$gb = $reader->getGbSequence();
echo $gb->getTopology();             // "LINEAR"
echo $gb->getDivision();             // "HUM"

foreach ($reader->getFeatures() as $feature) {
    echo $feature->getFtKey() . ": " . $feature->getFtQual()
        . " = " . $feature->getFtValue() . "\n";
}

foreach ($reader->getReferences() as $reference) {
    echo $reference->getPubmed() . " - " . $reference->getJournal() . "\
       n";
}
```

What this is actually useful for: the same use as GenBank (extracting sequence, organism, features, bibliography), applied to files coming from ENA; directly retrieving a reference's PubMed identifier, read from the `RX` line with no extra processing.

Persistence: `$recordManager->find('EMBL', $id)` retrieves an already-stored record; `storeFile($name, 'EMBL', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.4 PDB

A PDB record (Protein Data Bank) describes an experimentally resolved three-dimensional structure (X-ray crystallography, NMR…): the atomic coordinates of each residue, the organization into helices and sheets, and the chain or chains making up the structure. This is a very different format from the previous three — you're not reading a sequence annotation here, but a geometry.

Format to pass to `readDatabase()`: `'PDB'`

A record opens with a `HEADER` line and closes with an `END` line:

```
HEADER    CYTOKINE                                01-JAN-95   1TNF
TITLE     TUMOR NECROSIS FACTOR-ALPHA
COMPND    MOL_ID: 1; MOLECULE: TUMOR NECROSIS FACTOR; CHAIN: A, B, C;
SOURCE    MOL_ID: 1; ORGANISM_SCIENTIFIC: HOMO SAPIENS;
```

```
KEYWDS    CYTOKINE, INFLAMMATION
EXPDTA    X-RAY DIFFRACTION
AUTHOR    M.J.ECK,S.R.SPRANG
SEQRES   1 A  157  VAL ARG SER SER SER ARG THR PRO SER ASP LYS PRO VAL
HELIX    1   1 SER A  109  ASN A  113  1
                                      5
SHEET    1   A 4 THR A  77  MET A  83  0
CRYST1   66.100   66.100  116.700  90.00  90.00 120.00 P 32 2 1       6
ATOM      1  N   VAL A   1      33.611  33.560  20.041  1.00 25.00
             N
END
```

What the reader returns: no common objects here — PDB describes a geometry, not an annotated sequence, and therefore doesn't inherit from `ParseDbAbstractManager`.

Method	Content
`getIdCode()`	The 4-character structure code (e.g. `"1TNF"`)
`getClassification()` / `getTitle()` / `getDepositionDate()`	Classification, title and deposition date
`getCompounds()` / `getSources()`	Arrays of blocks (one per molecule) — `MOL_ID`, `MOLECULE`, `CHAIN` fields for the former, `ORGANISM_SCIENTIFIC` for the latter
`getKeywords()` / `getAuthors()`	Arrays of strings
`getExperimentalTechnique()`	E.g. `"X-RAY DIFFRACTION"`
`getSeqRes()`	Array `chain => sequence` — the protein sequence of each chain, reconstructed from the 3-letter codes
`getHelices()` / `getSheets()`	Arrays of `PdbHelix` / `PdbSheet` (section 2.4)
`getCryst1()`	Associative array — unit cell parameters (`a`, `b`, `c`, `alpha`, `beta`, `gamma`, `spaceGroup`, `z`)
`getAtoms()` / `getHetAtoms()`	Arrays of `PdbAtom` (section 2.4) — main-chain atoms / hetero-atoms (ligands, ions, water)

Warning

Only the record types most useful for structural bioinformatics are read (`HEADER`, `TITLE`, `COMPND`, `SOURCE`, `KEYWDS`, `EXPDTA`, `AUTHOR`, `SEQRES`, `HELIX`, `SHEET`, `CRYST1`, `ATOM`, `HETATM`). The many other record types found in the full PDB format (`CONECT`, `ANISOU`, `MASTER`...) are not handled.

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('1tnf.pdb');
$reader = DatabaseReaderFactory::readDatabase('PDB', $lines);

echo $reader->getIdCode();                  // "1TNF"
echo $reader->getExperimentalTechnique();   // "X-RAY DIFFRACTION"

// The sequence of each chain
```

```php
foreach ($reader->getSeqRes() as $chain => $sequence) {
    echo "Chain $chain: $sequence\n";
}

// The helices
foreach ($reader->getHelices() as $helix) {
    printf(
        "Helix %s: residues %d to %d\n",
        $helix->getHelixId(),
        $helix->getInitSeqNum(),
        $helix->getEndSeqNum()
    );
}

// Distance between two atoms (simple geometric calculation from the
// x, y, z coordinates)
$atoms = $reader->getAtoms();
$first = $atoms[0];
$last = end($atoms);
$distance = sqrt(
    ($first->getX() - $last->getX()) ** 2
    + ($first->getY() - $last->getY()) ** 2
    + ($first->getZ() - $last->getZ()) ** 2
);
```

What this is actually useful for: retrieving the sequence actually observed in the structure, chain by chain (to compare against the theoretical sequence from GenBank or Swiss-Prot); locating helices and sheets to understand the fold; using the atomic coordinates for geometric calculations (distances, contacts, a chain's center of mass); quickly finding the experimental technique used to judge how reliable the coordinates are.

Persistence: `$recordManager->find('PDB', $id)` retrieves an already-stored record; `storeFile($name, 'PDB', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.5 PROSITE

A PROSITE record does not describe a sequence but a *motif* (a "pattern", or a matrix) — the schema that characterizes a protein family or a functional site. It's the reference tool for quickly finding the signature that defines a known family and checking it against a sequence.

Format to pass to `readDatabase()`: `'PROSITE'`

A record opens with an `ID` line and closes with a `//` line:

```
ID   ATP_GTP_A; PATTERN.
AC   PS00017;
DT   APR-1990 (CREATED); DEC-2004 (DATA UPDATE); DEC-2004 (INFO UPDATE).
DE   ATP/GTP-binding site motif A (P-loop).
PA   [AG]-x(4)-G-K-[ST].
NR   /RELEASE=20.24,146629;   /TOTAL=602(578); /POSITIVE=583(559);
CC   /TAXO-RANGE=??????; /MAX-REPEAT=1;
3D   1YIC; 1TNF;
DR   P01375, TNFA_HUMAN, T; P00533, EGFR_HUMAN, T;
DO   PDOC00017;
//
```

What the reader returns: no common objects — PROSITE describes a motif, not an annotated sequence.

Method	Content
`getEntryName()` / `getEntryType()` / `getAccession()`	E.g. `"ATP_GTP_A"`, `"PATTERN"`, `"PS00017"`
`getDates()`	Array of event (`CREATED`, `DATA UPDATE`...) `=> date`
`getDescription()`	One-sentence description
`getPattern()`	The motif itself (`PA` field)
`getMatrix()`	`MA` field, kept as raw text
`getNumericalResults()` / `getComments()`	Associative arrays (`qualifier => value`), `NR` and `CC` fields
`getRule()`	`RU` field — additional rule, if any
`getPdbXrefs()`	Array of associated PDB codes (`3D` field)
`getDbRefs()`	Array of `PrositeDbRef` (section 2.4) — the Swiss-Prot proteins recorded as matching (or not) the motif
`getDocXref()`	Reference to the detailed documentation (`DO` field)

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('ps00017.prosite');
$reader = DatabaseReaderFactory::readDatabase('PROSITE', $lines);

echo $reader->getEntryName();  // "ATP_GTP_A"
echo $reader->getDescription();
echo $reader->getPattern();    // "[AG]-x(4)-G-K-[ST]."

// Separate true and false positives among the associated proteins
foreach ($reader->getDbRefs() as $ref) {
    if ($ref->isTruePositive()) {
        echo "True positive: " . $ref->getEntryName() . "\n";
    }
}
```

What this is actually useful for: getting a family's motif to search for it in your own sequence (with `SequenceMatchManager`, see further on); cross-referencing a motif with the PDB structures that illustrate it; separating, among the referenced proteins, those that truly match the motif from those listed for comparison (false positives).

Persistence: `$recordManager->find('PROSITE', $id)` retrieves an already-stored record; `storeFile($name, 'PROSITE', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.6 ExPASy ENZYME

An ExPASy ENZYME record describes an enzyme by its EC (*Enzyme Commission*) number — its catalyzed reaction, its cofactors, the diseases linked to a deficiency of that enzyme, and its cross-references to PROSITE and Swiss-Prot.

Format to pass to `readDatabase()`: `'EXPASY_ENZYME'`

A record opens with an `ID` line and closes with a `//` line:

```
ID   3.1.1.3
DE   Triacylglycerol lipase.
AN   Triglyceride lipase.
CA   (1) Triacylglycerol + H2O = diacylglycerol + a carboxylate.
CF   Ca(2+).
CC   -!- Wide specificity.
DI   Lipase deficiency, congenital; MIM: 246650.
PR   PROSITE; PDOC00110;
DR   P00590, LIP_HUMAN;
//
```

What the reader returns: no common objects here either — an enzyme described by its reaction is not an annotated sequence.

Method	Content
`getId()`	The EC number (e.g. `"3.1.1.3"`)
`getDescription()`	The enzyme's recommended name
`getAlternateNames()`	Array — synonyms (`AN` field, one per line)
`getCatalyticActivities()`	Array — one entry per catalyzed reaction (an enzyme with several substrates has several)
`getCofactors()`	Array (`CF` field)
`getComments()`	Free text (`CC` field)
`getDiseases()`	Array of `ExpasyDisease` (section 2.4)
`getPrositeRefs()`	Array of linked PROSITE documentation identifiers
`getSwissprotRefs()`	Array `accession => entry name` — the Swiss-Prot proteins carrying this activity

Warning

Don't confuse this reader with `RestrictionEnzymeManager` (section 2.6.4): ExPASy ENZYME covers metabolic enzymes in general (EC numbering), while `RestrictionEnzymeManager` deals specifically with restriction enzymes — those that cut DNA, already encountered in Chapter 1 with `EcoRI`. Two completely different databases, despite the similar name.

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('3.1.1.3.txt');
$reader = DatabaseReaderFactory::readDatabase('EXPASY_ENZYME', $lines);

echo $reader->getId();                    // "3.1.1.3"
echo $reader->getDescription();
print_r($reader->getCatalyticActivities());

foreach ($reader->getDiseases() as $disease) {
    echo $disease->getDisease() . " (" . $disease->getReference() . ")\n
        ";
}
```

What this is actually useful for: finding an enzyme by its EC number and reading its reaction; listing its required cofactors; tracing the diseases linked to a deficiency of this enzyme; finding the Swiss-Prot proteins that carry it, without having to search separately.

Persistence: `$recordManager->find('EXPASY_ENZYME', $id)` retrieves an already-stored record; `storeFile($name, 'EXPASY_ENZYME', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.7 PDBSTR

A PDBSTR record describes a member of a structural family derived from PDB — it's a grouping of similar structures, not the structure itself (for that, see `PDB` above).

Format to pass to `readDatabase()`: `'PDBSTR'`

A record fits on a single `MEMBER` line and closes with a `//` line:

```
MEMBER       1YIC_01        108     PROTEIN       1YIC   97/02/18   97/07/23
//
```

What the reader returns:

Method	Content
`getMemberId()`	Member identifier (e.g. `"1YIC_01"`)
`getLength()`	Length, in residues
`getMolType()`	Molecule type (e.g. `"PROTEIN"`)
`getEntryGroup()`	PDB code of the group the member belongs to
`getCreateDate()` / `getUpdDate()`	Creation and last-update dates

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('member.pdbstr');
$reader = DatabaseReaderFactory::readDatabase('PDBSTR', $lines);

echo $reader->getMemberId();     // "1YIC_01"
echo $reader->getEntryGroup();   // "1YIC"
echo $reader->getUpdDate();      // "97/07/23"
```

What this is actually useful for: grouping structures from the same family under a common group; tracking a member's last-update date to know whether a more recent structure exists.

Persistence: `$recordManager->find('PDBSTR', $id)` retrieves an already-stored record; `storeFile($name, 'PDBSTR', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.8 PRINTS

A PRINTS record describes a *fingerprint* of a protein family — a set of several conserved motifs which, taken together, identify the family more precisely than a single motif (like PROSITE's).

Format to pass to `readDatabase()`: `'PRINTS'`

A record opens with a `gc;` line:

```
gc; TNFALPHA
gn; TNFALPHA
ga; 16-NOV-1995; UPDATE 06-JUN-1999
gd; Tumor necrosis factor alpha fingerprint.
```

Warning

PRINTS has no end-of-record marker: a stream containing several of them cannot be split automatically, and reading stops at the end of the file. In practice, process one PRINTS file per call.

What the reader returns:

Method	Content
`getEntryName()` / `getEntryType()`	`gc;` and `gn;` fields
`getCreateDate()` / `getUpdDate()`	Read from the `ga;` field
`getDescription()`	`gd;` field

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('tnfalpha.prints');
$reader = DatabaseReaderFactory::readDatabase('PRINTS', $lines);

echo $reader->getEntryName();    // "TNFALPHA"
echo $reader->getDescription();
echo $reader->getUpdDate();      // "06-JUN-1999"
```

What this is actually useful for: quickly identifying a family fingerprint by its name and description; checking how fresh the entry is before trusting it.

Persistence: `$recordManager->find('PRINTS', $id)` retrieves an already-stored record; `storeFile($name, 'PRINTS', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.9 BLOCKS

A BLOCKS record describes a *block* — an ungapped conserved region of a protein family, positioned at a certain distance from the previous block of the same family.

Format to pass to `readDatabase()`: `'BLOCKS'`

A record opens with an `ID` line and closes with a `//` line:

```
ID   TNF_FAMILY; BLOCK
AC   IPB002128C; distance from previous block=(30,31)
DE   Tumor necrosis factor family signature
BL   THR; width=15; seqs=12; 99.5%;
//
```

What the reader returns:

Method	Content
`getId()` / `getAccession()`	Block name and accession
`getDistMin()` / `getDistMax()`	Minimum and maximum distance, in residues, from the family's previous block

Method	Content
`getDescription()`	`DE` field
`getAaTriplet()`	First element of the `BL` field (e.g. `"THR"`)

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('tnf_family.blocks');
$reader = DatabaseReaderFactory::readDatabase('BLOCKS', $lines);

echo $reader->getId();                          // "TNF_FAMILY"
printf("Distance from previous block: %d-%d\n", $reader->getDistMin(),
    $reader->getDistMax());
```

What this is actually useful for: locating a protein family's conserved blocks and their spacing, to reconstruct the family's overall organization.

Persistence: `$recordManager->find('BLOCKS', $id)` retrieves an already-stored record; `storeFile($name, 'BLOCKS', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.10 ProDom

A ProDom record describes a family of protein *domains* — a region found, repeated or not, across several different proteins.

Format to pass to `readDatabase()`: `'PRODOM'`

A record opens with an `ID` line and closes with a `//` line:

```
ID  20167  p2002.1                              10 seq.
AC  PD000001
KW    FADR(2)  Y586(1)  // COMPLETE  PROTEOME  DNA-BINDING  FATTY
//
```

What the reader returns:

Method	Content
`getEntryNo()` / `getAccession()`	Entry number and accession
`getRelease()`	Database release (the leading `"p"` is stripped)
`getDomainCount()`	Number of sequences in the family
`getFreqNames()`	Array `name => occurrences` — the most frequent protein names in the family
`getKeywords()`	Array — keywords (second half of the `KW` field, after the `//`)

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('pd000001.prodom');
$reader = DatabaseReaderFactory::readDatabase('PRODOM', $lines);

```

```php
echo $reader->getAccession();        // "PD000001"
echo $reader->getDomainCount();      // 10

// The most frequent protein names in the family
arsort($names = $reader->getFreqNames());
foreach ($names as $name => $occurrences) {
    echo "$name: $occurrences\n";
}
```

What this is actually useful for: counting a family's domains and seeing the most frequent protein names to quickly identify what it is; retrieving the associated keywords.

Persistence: `$recordManager->find('PRODOM', $id)` retrieves an already-stored record; `storeFile($name, 'PRODOM', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.11 AAINDEX

An AAINDEX (AAINDEX1) record describes a numerical physicochemical property of the twenty amino acids — hydrophobicity, volume, charge…— measured or computed in a given publication.

Format to pass to `readDatabase()`: `'AAINDEX'`

A record opens with an `H` line and closes with a `//` line:

```
H ARGP820101
D Hydrophobicity index (Argos et al., 1982)
A Argos, P., Rao, J.K.M. and Hargrave, P.K.
T Structural prediction of membrane-bound proteins
J Eur. J. Biochem. 128, 565-575 (1982)
R LIT:1810048b PMID:1575719
I     A/L     R/K     N/M     D/F     C/P     Q/S     E/T     G/W     H/Y
         I/V
     0.61    0.60    0.06    0.46    1.07    0.00    0.47    0.07    0.61
            2.22
//
```

What the reader returns:

Method	Content
`getAccession()`	Index identifier (`H` field)
`getDescription()`	`D` field
`getAuthor()` / `getTitle()` / `getJournal()`	Source publication (`A`, `T`, `J` fields)
`getLitRefs()`	Array `database => identifier` (`R` field, e.g. `"PMID" => "1575719"`)

Note

The numerical values of the index itself (`I` field, twenty values, one per amino acid) are not broken out: it's a fixed array of twenty numbers that the reader doesn't expose through a dedicated getter.

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('argp820101.aaindex');
$reader = DatabaseReaderFactory::readDatabase('AAINDEX', $lines);

echo $reader->getAccession();     // "ARGP820101"
echo $reader->getDescription();   // "Hydrophobicity index (Argos et al., 1982)"
echo $reader->getJournal();

print_r($reader->getLitRefs());   // ["LIT" => "1810048b", "PMID" => "1575719"]
```

What this is actually useful for: finding a physicochemical index of amino acids by its identifier; tracing back to the publication that measured or computed it, to check its methodology before using it in an analysis.

Persistence: `$recordManager->find('AAINDEX', $id)` retrieves an already-stored record; `storeFile($name, 'AAINDEX', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.12 PIR

A PIR (Protein Information Resource) record describes an annotated protein, in a particular grammar: each field qualifies its data with `#key value` pairs, rather than using a dedicated field per piece of information the way GenBank or Swiss-Prot do.

Format to pass to `readDatabase()`: `'PIR'`

A record opens with an `ENTRY` line and closes with a `//` line:

```
ENTRY            RHTDTO  #type complete
TITLE            tumor necrosis factor alpha precursor - human
ORGANISM         #formal_name Homo sapiens #common_name human
DATE             15-Jun-2001 #sequence_revision 15-Jun-2001 #text_change
    15-Jun-2001
ACCESSIONS       A12345; B67890
KEYWORDS         cytokine; glycoprotein
SUMMARY          #length 233  #molecular-weight 25644  #checksum 4657
//
```

What the reader returns:

Method	Content
`getEntryName()` / `getEntryType()`	Read from `ENTRY` (name, then the `#type` qualifier)
`getTitle()`	`TITLE` field
`getAccessions()`	Array, `ACCESSIONS` field
`getOrganism()` / `getSpecies()`	Common and scientific names (qualifiers `#common_name` / `#formal_name` of `ORGANISM`)
`getCreateDate()` / `getSeqrevDate()` / `getTxtchgDate()`	Three dates from the `DATE` field (creation, sequence revision, text change)

Method	Content
`getLength()` / `getMolwt()` / `getChecksum()`	Qualifiers of the `SUMMARY` field
`getKeywords()`	Array, `KEYWORDS` field

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('rhtdto.pir');
$reader = DatabaseReaderFactory::readDatabase('PIR', $lines);

echo $reader->getEntryName();   // "RHTDTO"
echo $reader->getOrganism();    // "human"
echo $reader->getLength();      // 233
echo $reader->getChecksum();    // "4657"
```

What this is actually useful for: reading a PIR entry despite its particular qualifier-based grammar; getting a protein's mass and length with no calculation needed; checking the integrity of a downloaded file via its checksum.

Persistence: `$recordManager->find('PIR', $id)` retrieves an already-stored record; `storeFile($name, 'PIR', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.13 PRF

A PRF (Protein Research Foundation, also called SEQDB) record describes a published protein: its sequence, its taxonomy, and the bibliographic reference that made it public.

Format to pass to `readDatabase()`: `'PRF'`

A record opens with a `CODE` line and closes with a `///` line (three bars, like PMD — not two like the GenBank family):

```
CODE        0901234A
NAME        TUMOR NECROSIS FACTOR ALPHA
SOURCE      HUMAN
cname       HOMO SAPIENS
taxon       EUKARYOTA; METAZOA; CHORDATA; MAMMALIA
JOURNAL     J. Biol. Chem.
AUTHOR      Eck,M.J., Sprang,S.R.
TITLE       The structure of TNF
KEYWORD     cytokine  inflammation
CROSSREF    PIR=ICHU2;PIR=ICGI2
SEQUENCE    MSTESMIRDV ELAEEALPKK TGGPQGSRRC
///
```

What the reader returns:

Method	Content
`getEntryCode()` / `getEntryName()`	`CODE` and `NAME` fields

Method	Content
`getSource()` / `getCommonName()`	`SOURCE` and `cname` fields
`getTaxonomy()`	Array, `taxon` field
`getJournal()` / `getAuthors()` / `getTitle()`	Bibliographic reference
`getKeywords()`	Array, `KEYWORD` field
`getComment()`	`COMMENT` field
`getCrossRefs()`	Array of pairs `["db" => ..., "id" => ...]`, `CROSSREF` field
`getSequence()`	The protein sequence itself

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('0901234a.prf');
$reader = DatabaseReaderFactory::readDatabase('PRF', $lines);

echo $reader->getEntryName();  // "TUMOR NECROSIS FACTOR ALPHA"
echo $reader->getSequence();
print_r($reader->getTaxonomy());

foreach ($reader->getCrossRefs() as $ref) {
    echo $ref['db'] . ": " . $ref['id'] . "\n";
}
```

What this is actually useful for: retrieving the sequence of a published protein, its taxonomy and its bibliographic reference; finding equivalent identifiers in other databases via the cross-references.

Persistence: `$recordManager->find('PRF', $id)` retrieves an already-stored record; `storeFile($name, 'PRF', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.14 PMD

A PMD (Protein Mutant Database) record describes a mutation reported in the literature for a given protein, along with the article that documents it.

Format to pass to `readDatabase()`: `'PMD'`

A record opens with an `ENTRY` line and closes with a `///` line (like PRF):

```
ENTRY           A000300 - Artificial                    2607383
AUTHORS         Eck,M.J. & Sprang,S.R.
MEDLINE         89123456
JOURNAL         J. Biol. Chem.
TITLE           Engineering TNF variants
///
```

What the reader returns:

Method	Content
`getEntryType()` / `getEntryNo()`	First character and next six of the `ENTRY` field
`getMutationType()` / `getArticleNo()`	Rest of the `ENTRY` field
`getAuthors()`	Array, `AUTHORS` field
`getMedlineNo()`	`MEDLINE` field
`getJournal()` / `getTitle()`	Bibliographic reference

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('a000300.pmd');
$reader = DatabaseReaderFactory::readDatabase('PMD', $lines);

echo $reader->getEntryNo();        // the entry number
echo $reader->getMutationType();
echo $reader->getTitle();
```

What this is actually useful for: finding the mutations described for a protein and the scientific article that reports them, for example for a structure-function study.

Persistence: `$recordManager->find('PMD', $id)` retrieves an already-stored record; `storeFile($name, 'PMD', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.15 ENTREZ

An Entrez record describes a genome in the NCBI Entrez system, with the same fixed header columns as GenBank (`LOCUS`, `DEFINITION`, `ACCESSION`...) but no `FEATURES` table or `ORIGIN` sequence: this is pure annotation.

Format to pass to `readDatabase()`: `'ENTREZ'`

A record opens with a `LOCUS` line and closes with a `//` line:

```
LOCUS       NC_001416              48502 bp    DNA     linear   PHG 01-
   JAN-2020
DEFINITION  Enterobacteria phage lambda, complete genome.
ACCESSION   NC_001416
VERSION     NC_001416.1  GI:9626243
KEYWORDS    complete genome.
SOURCE      Escherichia phage lambda
  ORGANISM  Escherichia phage lambda
            Viruses; Duplodnaviria; Heunggongvirae.
REFERENCE   1  (bases 1 to 48502)
  AUTHORS   Sanger,F., Coulson,A.R., Hong,G.F. and Petersen,G.B.
  TITLE     Nucleotide sequence of bacteriophage lambda DNA
  JOURNAL   J. Mol. Biol. 162:729-773(1982)
//
```

What the reader returns: not the common objects from section 2.3.3 (this reader doesn't inherit from `ParseDbAbstractManager`, having no `FEATURES` table) but a very similar set of methods:

Method	Content
`getEntryName()` / `getMolType()` / `getLength()`	Read from the `LOCUS` line
`getEntryDate()` / `getDivision()` / `getTopology()` / `getStrands()`	Same, other `LOCUS` columns
`getDefinition()`	`DEFINITION` field
`getPrimAcc()` / `getAccession()`	First and all accessions (`ACCESSION` field)
`getVersion()` / `getNcbiGiId()`	`VERSION` field
`getKeywords()`	Array, `KEYWORDS` field
`getSource()` / `getOrganism()` / `getTaxonomy()`	`SOURCE` and `ORGANISM` fields
`getReferences()`	Array of `EntrezReference` (section 2.4)

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('nc_001416.entrez');
$reader = DatabaseReaderFactory::readDatabase('ENTREZ', $lines);

echo $reader->getEntryName();    // "NC_001416"
echo $reader->getDefinition();
echo $reader->getLength();       // 48502

foreach ($reader->getReferences() as $reference) {
    echo $reference->getTitle() . " - " . $reference->getJournal() . "\n
        ";
}
```

Note

An Entrez file looks a lot like a GenBank file (same header columns), but the two readers return different objects: `ENTREZ` has no `FEATURES` table or sequence, whereas `GENBANK` (section 2.5.2) has both. Choose the format based on what your file actually contains, not just on its appearance.

What this is actually useful for: reading the annotation of an NCBI Entrez genomic record when the file carries no `FEATURES` table or sequence proper; quickly retrieving the definition, organism and bibliography.

Persistence: `$recordManager->find('ENTREZ', $id)` retrieves an already-stored record; `storeFile($name, 'ENTREZ', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.16 GENOME

A GENOME record summarizes the sequencing progress of an organism: genome size, number of associated GenBank entries, degree of completion, and the publications reporting on it.

Format to pass to `readDatabase()`: `'GENOME'`

A record opens with an `ORGANISM` line and closes with a `//` line. Continuation lines are indented with a tab:

```
ORGANISM        Escherichia coli
COMMON_NAME     E. coli
CLASSIFICATION  Bacteria; Proteobacteria; Gammaproteobacteria
COMPLETED       Yes
GB_RELEASE      250
GB_ENTRIES      12
GB_BASEPAIRS    4641652
GENOME_SIZE     4641652
REF_TYPE        Journal Article
REF_AUTHOR      Blattner,F.R.; Plunkett,G.
REF_TITLE       The complete genome sequence of Escherichia coli K-12
REF_JOURNAL     Science
REF_VOLUME      277
REF_PAGES       1453-1462
REF_YEAR        1997
//
```

What the reader returns:

Method	Content
`getOrganism()` / `getCommonName()`	Scientific and common names
`getTaxClass()`	Array, `CLASSIFICATION` field
`getIsComplete()`	`COMPLETED` field, as written
`getGbRelease()` / `getGbEntries()` / `getGbBasepairs()`	State of GenBank coverage
`getSize()`	Size of the haploid genome, in base pairs
`getReferences()`	Array of `GenomeReference` (section 2.4) — a record can have several

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('ecoli.genome');
$reader = DatabaseReaderFactory::readDatabase('GENOME', $lines);

echo $reader->getOrganism();      // "Escherichia coli"
echo $reader->getIsComplete();    // "Yes"
echo $reader->getSize();          // 4641652

foreach ($reader->getReferences() as $reference) {
    echo $reference->getTitle() . " (" . $reference->getYear() . ")\n";
}
```

What this is actually useful for: comparing the sequencing progress of several organisms; quickly retrieving the reference publication for a genome.

Persistence: `$recordManager->find('GENOME', $id)` retrieves an already-stored record; `storeFile($name, 'GENOME', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.17 HGBase

An HGBase record describes a biallelic human genetic variant (a point mutation) and its frequency in a given population.

Format to pass to `readDatabase()`: `'HGBASE'`

A record opens with a `HAPLOTYPEID` line and closes with a `//` line:

```
HAPLOTYPEID      HGVBASE:12345
ALLELE           A/G
ISINBLOCK        Yes
POPULATIONID     POP0042
POPULATION       Caucasian (USA) (216 individuals)
FREQUENCY        2% (1039 individuals)
SOURCEID         SRC0099
CITATION         Smith et al. 2001
SUBMITTER        Jan. W. Koper (SUB0001234)
SOURCECOMMENT    Genotyped by TaqMan assay
//
```

What the reader returns:

Method	Content
`getHaplotypeId()` / `getAllele()` / `getIsInBlock()`	Variant identification
`getPopulationId()` / `getPopName()` / `getPopIndiv()`	Observed population and its size
`getFreqPerc()` / `getFreqIndiv()`	Allele frequency, as a percentage and as a headcount
`getSourceId()` / `getCitation()`	Source reference
`getSubmitterName()` / `getSubmissionId()`	Who submitted the entry, and under what number
`getSourceComment()`	Free-text comment on the source

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('hgvbase_12345.txt');
$reader = DatabaseReaderFactory::readDatabase('HGBASE', $lines);

printf(
    "%s: frequency %.1f%% over %d individuals, population %s\n",
    $reader->getAllele(),
    $reader->getFreqPerc(),
    $reader->getFreqIndiv(),
    $reader->getPopName()
);
```

What this is actually useful for: reading a variant's allele frequencies by population; tracing back to the submitter and the genotyping method used.

Persistence: `$recordManager->find('HGBASE', $id)` retrieves an already-stored record; `storeFile($name, 'HGBASE', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.18 EPD

An EPD (Eukaryotic Promoter Database) record describes a eukaryotic promoter, in a two-letter tag grammar close to EMBL's.

Format to pass to `readDatabase()`: `'EPD'`

A record opens with an `ID` line and closes with a `//` line:

```
ID   HS_ACHA_1      standard; single; HUM.
AC   EP73001;
DT   15-JUN-2001 (REL. 75, CREATED)
DT   20-JUL-2010 (REL. 90, LAST SEQUENCE UPDATE)
DE   Human acetylcholine receptor alpha subunit promoter.
CC   TATA-less promoter.
//
```

What the reader returns:

Method	Content
`getEntryName()` / `getDataClass()` / `getInsiteType()` / `getTaxDiv()`	Read from the `ID` line
`getAccessions()`	Array, `AC` field
`getCreateDate()` / `getCreateRel()`	Creation date and release
`getSequpdDate()` / `getSequpdRel()`	Date and release of the last sequence update
`getNotupdDate()` / `getNotupdRel()`	Date and release of the last annotation update
`getDescription()` / `getComments()`	`DE` and `CC` fields

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('ep73001.epd');
$reader = DatabaseReaderFactory::readDatabase('EPD', $lines);

echo $reader->getEntryName();     // "HS_ACHA_1"
echo $reader->getDataClass();     // "standard"
echo $reader->getSequpdDate();    // "20-JUL-2010"
```

What this is actually useful for: finding a eukaryotic promoter, its class and its taxonomy; knowing how fresh it is before trusting it, as with Swiss-Prot.

Persistence: `$recordManager->find('EPD', $id)` retrieves an already-stored record; `storeFile($name, 'EPD', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.19 UniGene

A UniGene record groups together sequences believed to come from the same gene (a *cluster*): what tissues it's expressed in, and which proteins it resembles.

Format to pass to `readDatabase()`: `'UNIGENE'`

A record opens with an `ID` line and closes with a `//` line:

```
ID          Hs.1
TITLE       Tumor necrosis factor
EXPRESS     liver; kidney; brain
PROTSIM     ORG=Mus musculus; PROTGI=12345; PROTID=NP_001.1; PCT=87; ALN
   =200
SCOUNT      42
//
```

What the reader returns:

Method	Content
`getClusterId()` / `getTitle()`	Cluster identifier and title
`getExpression()`	Array of tissues (`EXPRESS` field)
`getProtSims()`	Array, one raw `PROTSIM` line per entry (protein similarities)
`getSeqCount()`	Number of sequences in the cluster (`SCOUNT` field)

Note

`PROTSIM` lines are kept as-is (raw text) rather than broken down field by field: their formatting varies from one file to the next.

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('hs.1.unigene');
$reader = DatabaseReaderFactory::readDatabase('UNIGENE', $lines);

echo $reader->getTitle();           // "Tumor necrosis factor"
print_r($reader->getExpression()); // ["liver", "kidney", "brain"]
echo $reader->getSeqCount();        // 42
```

What this is actually useful for: seeing in which tissues a gene cluster is expressed; knowing how many sequences make it up, and which proteins from other organisms it resembles.

Persistence: `$recordManager->find('UNIGENE', $id)` retrieves an already-stored record; `storeFile($name, 'UNIGENE', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.5.20 NCBI_LIT

An NCBI_LIT record describes a scientific journal referenced by the NCBI — its full title, its abbreviations, its ISSN/ESSN numbers.

Format to pass to `readDatabase()`: `'NCBI_LIT'`

A record opens with a `JrId:` line and closes with a line of dashes — the only format in the library that doesn't close on `//` or `///`:

```
JrId: 12345
JournalTitle: Journal of Biological Chemistry
MedAbbr: J Biol Chem
ISSN: 0021-9258
ESSN: 1083-351X
IsoAbbr: J. Biol. Chem.
NlmId: 2985121R
----------
```

What the reader returns:

Method	Content
`getId()`	`JrId` field
`getTitle()`	Full title of the journal
`getMedAbbr()` / `getIsoAbbr()`	Two different abbreviations (medical, ISO)
`getIssn()` / `getEssn()`	Print and electronic ISSN numbers
`getNlmId()`	National Library of Medicine identifier

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('journals.txt');
$reader = DatabaseReaderFactory::readDatabase('NCBI_LIT', $lines);

echo $reader->getTitle();     // "Journal of Biological Chemistry"
echo $reader->getMedAbbr();   // "J Biol Chem"
echo $reader->getIssn();      // "0021-9258"
```

What this is actually useful for: resolving a journal abbreviation encountered in a bibliographic reference (for example from a `getJournal()` field of GenBank or Swiss-Prot) into its full title and standardized identifiers.

2.5.21 The TRANSFAC family

TRANSFAC describes gene regulation: transcription factors, their DNA binding sites, the genes they regulate. The database publishes one file per type of information (`matrix.dat`, `gene.dat`, `class.dat`, `cell.dat`, `factor.dat`, `site.dat`), all written in the same grammar: a two-letter tag at the start of a line, its data starting at the fifth column, and a `//` line that closes the record. BioPHP therefore gives this family one reader per file, all built on a shared base: each already knows the accession (`getAccession()`), the identifier (`getId()`) and the two creation and update dates (`getDateCreated()`/`getDateUpdated()`) — only the fields specific to each file need to be discovered card by card.

Persistence: `$recordManager->find('NCBI_LIT', $id)` retrieves an already-stored record; `storeFile($name, 'NCBI_LIT', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

TRANSFAC_MATRIX

Describes the position weight matrix of a transcription factor — for each position of the binding site, how many times each nucleotide (A, C, G, T) was observed.

Format: `'TRANSFAC_MATRIX'` — opens with `AC`, closes with `//`.

```
AC  M00001
ID  V$MYOD_01
DT  20.06.1990 (created); ewi.
NA  MyoD
DE  Myoblast determination protein
01      1       2       2       0       N
02      0       0       4       0       C
BA  15 sequences
CC  quality: A
//
```

Method	Content
`getName()`	`NA` field
`getDescription()`	`DE` field
`getMatrix()`	Array — one row per position, each `["A"=>..., "C"=>..., "G"=>..., "T"=>..., "consensus"=>...]`
`getBasis()`	`BA` field — basis of the matrix's computation
`getComments()`	`CC` field

```php
<?php
use Amelaye\BioPHP\Domain\Database\Factory\DatabaseReaderFactory;

$lines = file('m00001.transfac');
$reader = DatabaseReaderFactory::readDatabase('TRANSFAC_MATRIX', $lines)
    ;

echo $reader->getName();  // "MyoD"
foreach ($reader->getMatrix() as $position => $row) {
    printf("Position %d: A=%d C=%d G=%d T=%d\n",
        $position, $row['A'], $row['C'], $row['G'], $row['T']);
}
```

What this is useful for: getting a position weight matrix to search for potential binding sites in a sequence (sliding-window scan, outside BioPHP).

Persistence: `$recordManager->find('TRANSFAC_MATRIX', $id)` retrieves an already-stored record; `storeFile($name, 'TRANSFAC_MATRIX', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

TRANSFAC_GENE

Describes a gene whose regulation TRANSFAC documents.

Format: `'TRANSFAC_GENE'` — opens with `AC`, closes with `//`.

```
AC  G000123
ID  G_TNF
SD  TNF
DE  Tumor necrosis factor gene
```

```
OS   human, homo sapiens
OC   Eukaryota; Metazoa; Chordata; Mammalia
CO   CE0000123
//
```

Method	Content
`getShortDescription()` / `getDescription()`	`SD` and `DE` fields
`getOrganism()` / `getSpecies()`	Common and scientific names (`OS` field)
`getTaxClass()`	Array, `OC` field
`getCompelAccessions()`	Array of linked COMPEL accessions (composite elements, `CO` field)

```php
<?php
$reader = DatabaseReaderFactory::readDatabase('TRANSFAC_GENE', file('g000123.transfac'));
echo $reader->getShortDescription();   // "TNF"
echo $reader->getOrganism();           // "human"
```

What this is useful for: quickly identifying the gene regulated behind a TRANSFAC identifier, and its taxonomic classification.

Persistence: `$recordManager->find('TRANSFAC_GENE', $id)` retrieves an already-stored record; `storeFile($name, 'TRANSFAC_GENE', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

TRANSFAC_CLASS

Describes a structural class of transcription factors (zinc fingers, leucine zippers, helix-turn-helix…) and lists the factors that belong to it.

Format: `'TRANSFAC_CLASS'` — opens with `AC`, closes with `//`.

```
AC   C0001
ID   Basic domains
CL   1; Basic domains; 1.1; Leucine zipper factors
SD   bZIP transcription factors
BF   T00001 T00002 T00003
//
```

Method	Content
`getClassification()`	Array — the classification, from the broadest level to the most specific (`CL` field)
`getStructuralDescription()`	`SD` field
`getMemberFactors()`	Array of member factor identifiers (`BF` field)
`getComments()`	`CC` field

```php
<?php
$reader = DatabaseReaderFactory::readDatabase('TRANSFAC_CLASS', file('c0001.transfac'));
```

```php
foreach ($reader->getMemberFactors() as $factor) {
    echo $factor . "\n";
}
```

What this is useful for: placing a factor within its structural class; listing every factor that shares the same DNA-binding domain family.

Persistence: `$recordManager->find('TRANSFAC_CLASS', $id)` retrieves an already-stored record; `storeFile($name, 'TRANSFAC_CLASS', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

TRANSFAC_CELL

Describes the cell type or line that a transcription factor comes from — linking a binding observation to the tissue it was made in.

Format: `'TRANSFAC_CELL'` — opens with `AC`, closes with `//`.

```
AC  T00012
ID  HeLa
OS  human
SO  cervical carcinoma
CD  Immortalized cervical cancer cell line
//
```

Method	Content
`getOrganism()`	`OS` field
`getFactorSource()`	`SO` field — origin of the factor
`getDescription()`	`CD` field

```php
<?php
$reader = DatabaseReaderFactory::readDatabase('TRANSFAC_CELL', file('
    t00012.transfac'));
echo $reader->getOrganism() . " - " . $reader->getFactorSource();
```

What this is useful for: knowing which cell type an observed factor comes from, in order to correctly interpret a binding site described in TRANSFAC_SITE.

Persistence: `$recordManager->find('TRANSFAC_CELL', $id)` retrieves an already-stored record; `storeFile($name, 'TRANSFAC_CELL', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

TRANSFAC_FACTOR

The most complete card in the family: describes a transcription factor — its name, its synonyms, the organism it comes from, its structural class and its homologs in other species.

Format: `'TRANSFAC_FACTOR'` — opens with `AC`, closes with `//`.

```
AC  T00001
ID  MyoD
FA  MyoD1
SY  bHLHc1; MYOD1
OS  human, homo sapiens
OC  Eukaryota; Metazoa; Chordata; Mammalia
```

```
HO   mouse MyoD, rat MyoD
CL   C0001; Basic domains; 1.1
SQ   MELLSPPLRDVDLTAPDGSLCSFATTDDFYD
//
```

Method	Content
`getFactorName()`	`FA` field
`getSynonyms()`	Array, `SY` field
`getOrganism()` / `getSpecies()`	`OS` field
`getTaxClass()`	Array, `OC` field
`getHomologs()`	Array, `HO` field
`getClassAccession()` / `getClassId()` / `getClassDecimalNo()`	Reference to the structural class (`CL` field, to be matched against TRANSFAC_CLASS)
`getSequence()`	`SQ` field — factor sequence, if provided

```php
<?php
$reader = DatabaseReaderFactory::readDatabase('TRANSFAC_FACTOR', file('
    t00001.transfac'));
echo $reader->getFactorName();  // "MyoD1"
print_r($reader->getSynonyms());
print_r($reader->getHomologs());
```

What this is useful for: getting a transcription factor's card, its synonyms (useful for cross-referencing another database) and its homologs in other species.

Persistence: `$recordManager->find('TRANSFAC_FACTOR', $id)` retrieves an already-stored record; `storeFile($name, 'TRANSFAC_FACTOR', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

TRANSFAC_SITE

Describes an observed binding site — the portion of DNA recognized by a factor, its position relative to the start of the gene's transcription, and the experimental method that demonstrated it.

Format: `'TRANSFAC_SITE'` — opens with `AC`, closes with `//`.

```
AC   R00001
ID   MyoD_site_1
TY   D
DE   MyoD binding site in the myogenin promoter
RE   myogenin promoter
SQ   CACCTGT
SF   -120
BF   T00001
OS   human
MM   gel shift assay
//
```

Method	Content
`getSeqType()`	`TY` field (e.g. `"D"` for a DNA element derived from a gene)
`getDescription()` / `getGeneRegion()`	`DE` and `RE` fields
`getSequence()`	`SQ` field
`getFirstPosition()`	`SF` field — position, counted from the transcription start site (negative if upstream)
`getBindingFactors()`	Array of the factors that bind there (`BF` field, references TRANSFAC_FACTOR)
`getOrganism()` / `getMethod()` / `getComments()`	`OS`, `MM`, `CC` fields

```php
<?php
$reader = DatabaseReaderFactory::readDatabase('TRANSFAC_SITE', file('
    r00001.transfac'));
printf(
    "%s: position %s, factors: %s\n",
    $reader->getSequence(),
    $reader->getFirstPosition(),
    implode(', ', $reader->getBindingFactors())
);
```

What this is useful for: listing the known binding sites around a gene, the factors involved and the method that demonstrated them — for example, to build a regulation map of a promoter.

2.5.22 The KEGG family

KEGG describes metabolism: metabolic pathways, molecules, reactions, genes, genomes. Like TRANSFAC, it publishes one file per type of information, all written in the same grammar: a twelve-column tag, its data starting at column thirteen, and a `///` line (three bars, not two) that closes the record. All readers in this family already know the identifier (`getEntry()`) and the names/synonyms (`getNames()`) — the fields specific to each file need to be discovered card by card.

Persistence: `$recordManager->find('TRANSFAC_SITE', $id)` retrieves an already-stored record; `storeFile($name, 'TRANSFAC_SITE', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

KEGG_COMPOUND

Describes a chemical compound in metabolism: its formula, the reactions it takes part in, the enzymes that act on it, the metabolic pathways it belongs to.

Format: `'KEGG_COMPOUND'` — opens with `ENTRY`, closes with `///`.

```
ENTRY       C00031                          Compound
NAME        D-Glucose; Grape sugar; Dextrose
FORMULA     C6H12O6
REACTION    R00299 R00301 R01555
ENZYME      1.1.1.47  2.7.1.1  5.3.1.9
PATHWAY     PATH: map00010  Glycolysis / Gluconeogenesis
DBLINKS     CAS: 50-99-7
```

```
///
```

Method	Content
`getFormula()`	`FORMULA` field
`getReactions()`	Array of reaction identifiers (`REACTION` field)
`getEnzymes()`	Array of EC numbers (`ENZYME` field)
`getPathways()`	Array of pairs `[identifier, name]` (`PATHWAY` field)
`getDbLinks()`	Array of pairs `[database, identifier]` (`DBLINKS` field)

```php
<?php
$reader = DatabaseReaderFactory::readDatabase('KEGG_COMPOUND', file('
    c00031.kegg'));
echo $reader->getFormula();   // "C6H1206"
foreach ($reader->getPathways() as [$id, $name]) {
    echo "$id: $name\n";
}
```

What this is useful for: starting from a molecule and tracing back to the reactions and metabolic pathways involving it, without having to already know its formula.

Persistence: `$recordManager->find('KEGG_COMPOUND', $id)` retrieves an already-stored record; `storeFile($name, 'KEGG_COMPOUND', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

KEGG_REACTION

Describes a biochemical reaction: its equation (written with compound identifiers) and the enzymes that catalyze it.

Format: `'KEGG_REACTION'` — opens with `ENTRY`, closes with `///`.

```
ENTRY       R00299                      Reaction
NAME        ATP:D-glucose 6-phosphotransferase
DEFINITION  ATP + D-Glucose <=> ADP + D-Glucose 6-phosphate
EQUATION    C00002 + C00031 <=> C00008 + C00092
ENZYME      2.7.1.1
///
```

Method	Content
`getDefinition()`	The reaction written with compound names
`getEquation()`	The reaction written with compound identifiers
`getEnzymes()`	Array of EC numbers catalyzing the reaction
`getPathways()`	Array of pairs `[identifier, name]`

```php
<?php
$reader = DatabaseReaderFactory::readDatabase('KEGG_REACTION', file('
    r00299.kegg'));
echo $reader->getEquation();
print_r($reader->getEnzymes());
```

What this is useful for: reading a reaction's equation and identifying the enzymes that catalyze it — useful for reconstructing a metabolic pathway step by step.

Persistence: `$recordManager->find('KEGG_REACTION', $id)` retrieves an already-stored record; `storeFile($name, 'KEGG_REACTION', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

KEGG_ENZYME

The richest card in the family: brings together everything KEGG knows about an EC number — classification, substrates, products, genes encoding the enzyme, related diseases, motifs and known structures.

Format: `'KEGG_ENZYME'` — opens with `ENTRY`, closes with `///`.

```
ENTRY       EC 2.7.1.1                  Enzyme
NAME        hexokinase
CLASS       Transferases; Transferring phosphorus-containing groups
SYSNAME     ATP:D-hexose 6-phosphotransferase
REACTION    R00299
SUBSTRATE   ATP; D-glucose
PRODUCT     ADP; D-glucose 6-phosphate
GENES       HSA: 3098 3099 3101
DISEASE     H00409  Hexokinase deficiency
STRUCTURES  PDB: 1HKB 1HKC 1IG8
DBLINKS     ExplorEnz: 2.7.1.1
///
```

Method	Content
`getClassification()`	Array, `CLASS` field
`getSysname()`	The enzyme's systematic name
`getReactions()` / `getSubstrates()` / `getProducts()`	What the enzyme catalyzes
`getGenes()`	Array, `GENES` field (one item per organism)
`getDiseases()`	Array, `DISEASE` field
`getMotifs()`	Array, `MOTIF` field
`getStructures()`	Array of linked PDB identifiers
`getComment()` / `getPathways()` / `getOrthologs()` / `getDbLinks()`	Additional information

```php
<?php
$reader = DatabaseReaderFactory::readDatabase('KEGG_ENZYME', file('2.7.1.1.kegg'));
echo $reader->getSysname();
print_r($reader->getSubstrates());
print_r($reader->getDiseases());
print_r($reader->getStructures());  // PDB codes, to go read with the PDB parser
```

What this is useful for: getting an enzyme's complete card in a single call — substrates, products, genes, diseases, structures — and following up, for example, with a PDB read of the listed structures.

Persistence: `$recordManager->find('KEGG_ENZYME', $id)` retrieves an already-stored record; `storeFile($name, 'KEGG_ENZYME', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

KEGG_ORTHOLOG

Groups together genes from different species that descend from the same ancestral gene and perform the same function — which lets you transpose a metabolic pathway from one species to another.

Format: `'KEGG_ORTHOLOG'` — opens with `ENTRY`, closes with `///`.

```
ENTRY       K00844                      KO
NAME        HK; hexokinase
DEFINITION  hexokinase [EC:2.7.1.1]
CLASS       Metabolism; Carbohydrate metabolism; Glycolysis
GENES       HSA: 3098 3099 3101
            MMU: 15275 15277
DBLINKS     GO: 0004396
///
```

Method	Content
`getDefinition()`	`DEFINITION` field
`getClassification()`	Array, `CLASS` field
`getGenes()`	Array, one entry per organism (e.g. `"HSA: 3098 3099 3101"`)
`getDbLinks()`	Array of pairs `[database, identifier]`

```php
<?php
$reader = DatabaseReaderFactory::readDatabase('KEGG_ORTHOLOG', file('
    k00844.kegg'));
foreach ($reader->getGenes() as $line) {
    echo $line . "\n";  // "HSA: 3098 3099 3101", "MMU: 15275 15277"...
}
```

What this is useful for: comparing a group of orthologous genes across species — for example to check that a human gene has a known equivalent in mouse before designing an experiment.

Persistence: `$recordManager->find('KEGG_ORTHOLOG', $id)` retrieves an already-stored record; `storeFile($name, 'KEGG_ORTHOLOG', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

KEGG_GENOME

Names a sequenced organism for which KEGG maintains metabolic pathways, and links it to NCBI taxonomy.

Format: `'KEGG_GENOME'` — opens with `ENTRY`, closes with `///`.

```
ENTRY       T01001                      Genome
NAME        hsa, Homo sapiens (human)
```

```
DEFINITION  Homo sapiens (human)
TAXONOMY    TAX:9606
LINEAGE     Eukaryota; Metazoa; Chordata; Mammalia; Primates
///
```

Method	Content
`getDefinition()`	`DEFINITION` field
`getTaxonomy()`	NCBI taxonomy identifier, without the `TAX:` prefix
`getLineage()`	Array, `LINEAGE` field

```php
<?php
$reader = DatabaseReaderFactory::readDatabase('KEGG_GENOME', file('
    t01001.kegg'));
echo $reader->getDefinition();  // "Homo sapiens (human)"
echo $reader->getTaxonomy();    // "9606"
```

Note

Don't confuse this reader with `GENOME` (section 2.5.16): `KEGG_GENOME` identifies an organism for KEGG's "metabolism" view (pathways, taxonomy), whereas `GENOME` summarizes an organism's sequencing progress on the GenBank side. Two different pieces of information about the same organism, in two independent databases.

What this is useful for: placing a genome within its taxonomic lineage before looking up its metabolic pathways or orthologous groups.

Persistence: `$recordManager->find('KEGG_GENOME', $id)` retrieves an already-stored record; `storeFile($name, 'KEGG_GENOME', ...$files)` imports an entire file into the `parsed_record` table (§2.3.2).

2.6 Manipulating sequences

2.6.1 The main service: SequenceManager

This is the service you'll reach for most often: it groups together all the common operations on a DNA/RNA/protein sequence. The following table serves as an index by need.

Need	Methods
Transform	`complement()`, `halfSequence()`, `expandNa()` (expands ambiguous IUPAC symbols), `subSeq()`
Translate	`translate()`, `translateCodon()`, `getCodon()`, `countCodons()`
Search for a motif	`patPos()`, `patPoso()`, `patFreq()`, `findPattern()`, `symFreq()`
Physicochemical properties	`molwt()` (molecular mass), `charge()`, `chemicalGroup()`
Palindromes and mirrors	`isMirror()`, `findMirror()`, `isPalindrome()`, `findPalindrome()`

Example: translating a coding sequence into a protein (the same `$sequenceManager` as in the "vanilla" installation from the introduction):

```php
<?php
$protein = $sequenceManager->translate(
    "ATGGCCATTGTAATGGGCCGCTGA",
    0,  // reading frame: 0, 1 or 2
    1   // format: 1 = single-letter codes (M), 3 = three-letter codes (Met)
);

echo $protein; // e.g. "MAIVMGR*"
```

Note

`translate()` and the other methods that handle amino acids query BioAPI behind the scenes (via the adapters seen in Chapter 1) to know the codon table — which is why `SequenceManager` is built with the three adapters `AminoApi`, `NucleotidApi` and `ElementApi`, as shown in the introduction.

`SequenceBuilder` is a thin layer around `SequenceManager`: it holds on to the current sequence (built from a `Sequence` object, typically the one returned by a reader from the previous section) to avoid passing it as an argument on every call. All the methods above also exist on it, without the first `$sequence` argument.

2.6.2 Sequences that validate themselves

In addition, BioPHP offers a few small immutable objects that wrap a string and check, as soon as they're created, that it contains only symbols valid for their alphabet: `DnaSequence`, `RnaSequence`, `AminoAcidSequence`.

```php
<?php
use Amelaye\BioPHP\Domain\Sequence\ValueObject\DnaSequence;

$dna = new DnaSequence("ATGGCCATTGTAATGGGCCGCTGA");
echo $dna->getLength();
echo $dna->reverse();
$rna = $dna->toRna(); // DNA -> RNA conversion

$protein = new \Amelaye\BioPHP\Domain\Sequence\ValueObject\
    AminoAcidSequence("MAIVMGR*");
var_dump($protein->hasStop());          // true: contains a stop codon
echo $protein->truncateAtStop();        // "MAIVMGR", cut at the stop
```

`MolecularSequenceFactory` knows how to automatically build the right object (`DnaSequence`, `RnaSequence`...) from a sequence coming out of a reader from the previous section, by looking at the molecule type it read (`fromEntity()`).

2.6.3 Proteins: ProteinManager

A deliberately lightweight service, for the two most commonly requested pieces of information about a protein: its length (`seqlen()`) and its molecular mass (`molwt()`).

2.6.4 Restriction enzymes: RestrictionEnzymeManager

We can now redo the "EcoRI" example from Chapter 1 properly, without building the motif search ourselves:

```php
<?php
// $typeIIEndonucleaseApi: the Chapter 1 adapter (Api\
    TypeIIEndonucleaseApi)
$manager = new \Amelaye\BioPHP\Domain\Sequence\Service\
    RestrictionEnzymeManager(
     $typeIIEndonucleaseApi,
     new \Amelaye\BioPHP\Domain\Sequence\Entity\Enzyme()
);

$manager->parseEnzyme("EcoRI");
$manager->setSequenceManager($sequenceManagerOfTheSequence);
$fragments = $manager->cutSeq();

print_r($fragments); // the DNA pieces obtained after cutting
```

`findRestEn()` does the reverse: you give it a motif (and, optionally, a cut position or a motif length), and it returns the known enzymes that match — handy when you've identified a cut site but don't yet know which enzyme produced it.

2.6.5 Aligning sequences: SequenceAlignmentManager

Reads a multiple alignment file already produced by a tool like ClustalW (FASTA and CLUSTAL formats supported), then lets you explore it:

```php
<?php
$alignment = new \Amelaye\BioPHP\Domain\Sequence\Service\
    SequenceAlignmentManager($sequenceBuilder);
$alignment->setFilename('my_alignment.aln');
$alignment->setFormat('CLUSTAL');
$alignment->parseFile();

echo $alignment->consensus(80);  // consensus sequence (80% threshold)
print_r($alignment->resVar(80)); // variable positions below this
    threshold
```

2.6.6 Comparing two sequences: SequenceMatchManager

For measuring similarity between two sequences: Hamming distance (`hamdist()`, sequences of equal length), Levenshtein distance (`levdist()`, tolerates insertions/deletions), or a letter-by-letter comparison driven by a substitution matrix (`match()`, `setSubMatrix()`) — the same idea as the PAM250 matrix exposed by BioAPI in Chapter 1.

2.7 The toolbox

A few more modest functions, grouped by theme.

Class	What it offers
`GeneticsFunctions`	`CountACGT()` (counts the bases of a sequence), `CountCG()`, `CreateInversion()` (reverse complementary strand), `RemoveNonCodingProt()`
`Mathematics Functions`	`Mean()`, `Median()`, `Variance()` — simple statistics over an array of values

Class	What it offers
`OligosManager`	`findOligos($sequence, $length)` counts the frequency of oligonucleotides of a given length in a sequence; `findZScore()` computes z-scores to compare these frequencies

2.8 Adapters towards BioAPI

Finally, the `Api/` folder contains the 14 classes that fetch reference data from BioAPI — one per resource from Chapter 1 (`AminoApi`, `NucleotidApi`, `ElementApi`, `TypeIIEndonucleaseApi`…). These are what feed `SequenceManager` and `RestrictionEnzymeManager`, presented in this chapter, behind the scenes. They work the same way already shown in the introduction: you give them an HTTP client and a serializer, and they return simple objects (one per row of data) rather than the API's raw JSON.

Chapter 3

BioTools

3.1 Overview

Where BioPHP (Chapter 2) gives you low-level building blocks — manipulating a sequence, reading a SWISSPROT file, calling the reference API — BioTools gives you **19 complete tools**, each one ready to become a page in your application : an already-validated input form, and the biological computation behind it. These are the well-known biophp.org mini-tools (DNA/protein translation, restriction enzyme digestion, melting temperature, palindrome search, chaos game representation, etc.), rewritten as Symfony components.

Note

Each tool is made of two classes : a `FormType` (in `Form/`) that describes the input fields and their constraints, and a `Manager` (in `Service/`) that contains the actual computation, independent of Symfony. The table in §3.6 maps all 19 pairs.

The point that structures this whole chapter : BioTools provides **neither a controller nor a template**. This is a deliberate choice — you are not forced to adopt the routes, the HTML layout or the page structure of biophp.org, you plug each form in wherever it suits you in your own application. The trade-off is that for each of the 19 tools, you have to write the controller action and the view that displays it yourself. This is exactly what each card in this chapter shows you : a complete Symfony implementation (modeled on the one from the demo site), and a "vanilla " variant for those not using Symfony.

Warning

Most managers need the API adapters seen in Chapter 1 (`NucleotidApiAdapter`, `AminoApiAdapter`, `VendorApiAdapter`...) injected into their constructor. In Symfony, autowiring takes care of this; in "vanilla ", you build them by hand. In both cases, **network access to the reference API is required** at call time — this is the first thing to check if a tool fails for no apparent reason.

3.2 Installation and configuration

3.2.1 Installation

```
composer require amelaye/biotools
```

BioTools depends on BioPHP, which in turn depends on two third-party bundles. In Symfony, the three bundles are registered in `config/bundles.php` :

```
return [
    // ...
    JMS\SerializerBundle\JMSSerializerBundle::class => ['all' => true],
    Amelaye\BioPHP\AmelayeBioPHPBundle::class => ['all' => true],
    Amelaye\BioTools\AmelayeBioToolsBundle::class => ['all' => true],
];
```

Note

In a "vanilla " install (without Symfony), there is no bundle to register : you instantiate the classes from `Form/` and `Service/` you need directly, as shown in each card. The Symfony form (`FormType`) then no longer plays any role — it is your own code that reads the received fields and passes them to the manager.

3.2.2 Configuration

The bundle works with no configuration at all — sensible default values are provided. Two settings can be overridden under the `amelaye_biotools` key :

```
# config/packages/amelaye_biotools.yaml
amelaye_biotools:
    nucleotids_graphs:
        path_graphs: 'created_files/'   # folder where generated SVGs
           are written
    protein_colors:
        # color palettes for reduced protein alphabets
```

Warning

`path_graphs` must exist and be writable by the PHP process, **and** be served publicly if you intend to display the generated images on a web page. This is a configuration point that is easy to forget on the first deployment — a 500 error when submitting one of the four tools that draw a chart (§3.5) is often the symptom.

3.3 The common pattern

All 19 tools follow the same pattern, which is simpler to understand once and for all rather than rediscovering it with every card :

1. The `FormType` describes the fields (often with a sample default value) and their validation constraints.
2. The controller creates the form, calls `handleRequest()`, and if the submission is valid, retrieves the data with `$form->getData()`.
3. This data is passed as-is to the main method of the corresponding `Manager`, which performs the computation and returns a structured result (an array, a formatted string, or the path of a generated SVG file).
4. The template displays this result.

3.3.1 The two home-grown validation constraints

Two Symfony `Constraint` classes recur across several forms, in `Validator/` :

Constraint	What it checks
`SequenceRecognition`	The string contains only valid NC-IUBMB codes (bases A/C/G/T, proteins, or degenerate nucleotides Y/R/W/S/K/M/D/V/H/B/N) once non-coding characters have been stripped. Rejects an empty sequence or one containing an unknown character.
`MeltingTemperature`	The same composition check, plus a length constraint : between 6 and 50 base pairs (only enforced if the field is not empty).

3.3.2 The Twig extension

Two tools (DNA/protein translation and sequence alignment) expose their formatting through a Twig extension, `BioToolsExtension`, rather than recomputing it in the controller :

```
{{ sequence_alignee|compare_alignment(autre_sequence) }}
{{ show_translations_aligned(sequence, frame) }}
{{ show_translations_aligned_complementary(frame) }}
```

Note

This filter and these two functions call `SequenceAlignmentManager::compareAlignment()` and `DnaToProteinManager::showTranslationsAligned(Complementary)()` respectively. You can just as well call these methods directly from the controller if you are not using Twig — this is what the corresponding cards show.

3.4 The 19 tool cards

Each card follows the same template : what the tool is for, a screenshot of the form, a table of fields (with their French translation, since the interface labels are in English), a screenshot of the result, the Symfony implementation followed by the "vanilla " one, and a callout box where relevant.

3.4.1 Chaos Game Representation (CGR and FCGR)

Described by Jeffrey (1990). Starting from the center of a square whose each vertex represents a base (A, C, G, T), each base of the sequence moves the current point halfway to its vertex, and this new position is plotted. After a large number of iterations, fractal patterns characteristic of the sequence emerge. The **FCGR** variant, described by Deschavanne *et al.* (1999), does not plot successive positions but rather **oligonucleotide frequencies** as a checkerboard.

Note

CGR and FCGR share the same `FormType` (`ChaosGameRepresentationType`) and the same manager. Only the fields displayed by the template differ : CGR only needs the name, the sequence and the image size; FCGR adds the choice of oligonucleotide length and number of strands.

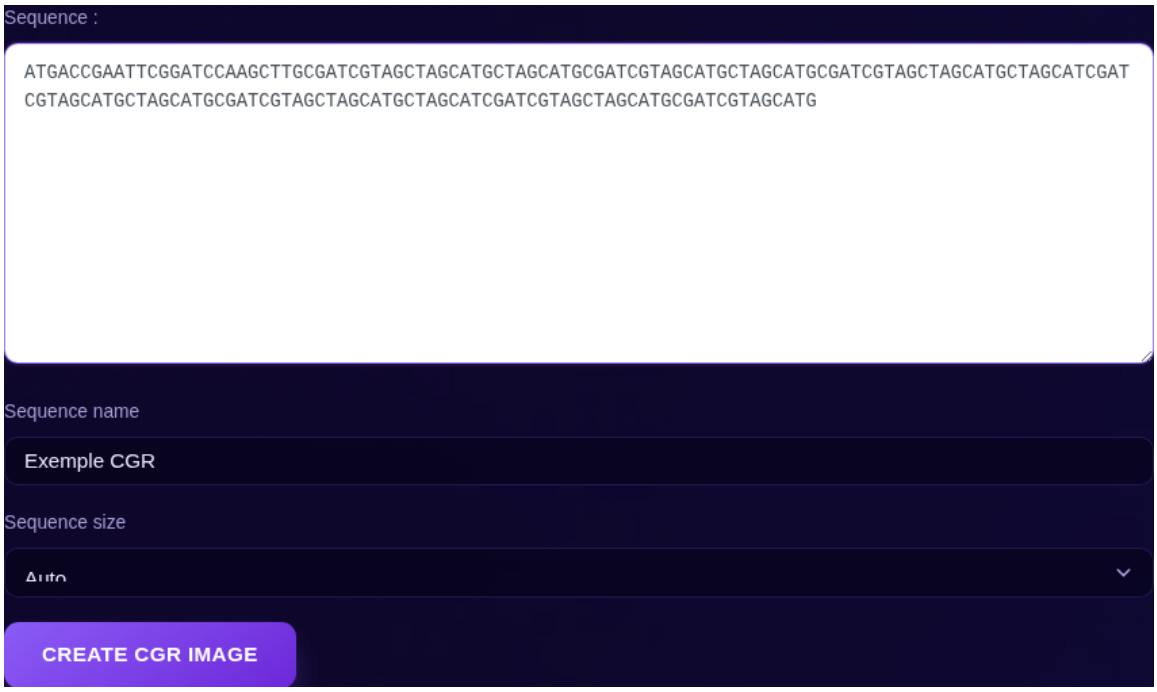

Figure 3.1: CGR form.

Field	**Label / translation**	**Type**	**Values, default, role**
`seq_name`	Sequence name *Sequence name*	text	Free text; used as the title of the generated image.
`seq`	Sequence *Sequence*	long text	DNA only; between 50 and 5,000,000 bp. A sample DNA sequence is provided by default.
`size`	Sequence size *Image size*	choice	`auto` / 1024×1024 / 512×512 / 256×256.

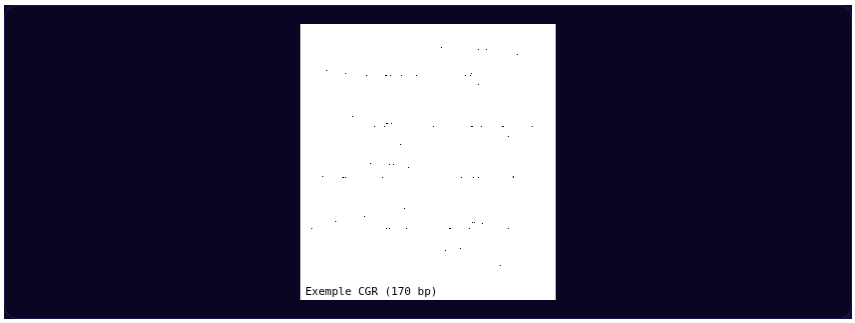

Figure 3.2: CGR result : each successive position of the sequence plotted in the square.

For FCGR, two additional fields appear :

Field	Label / translation	Type	Values, default, role
`len`	Search oligos of length *Oligonucleotide length*	choice	2 to 7.
`s`	Compute data for *Compute on*	choice	`2` = both strands (default) / `1` = upper strand only.
`map`	Show as image map *Show as clickable map*	checkbox	Unchecked by default; not recommended for long oligonucleotides.
`freq`	Show oligonucleotide frequencies *Show frequencies*	checkbox	Checked by default.

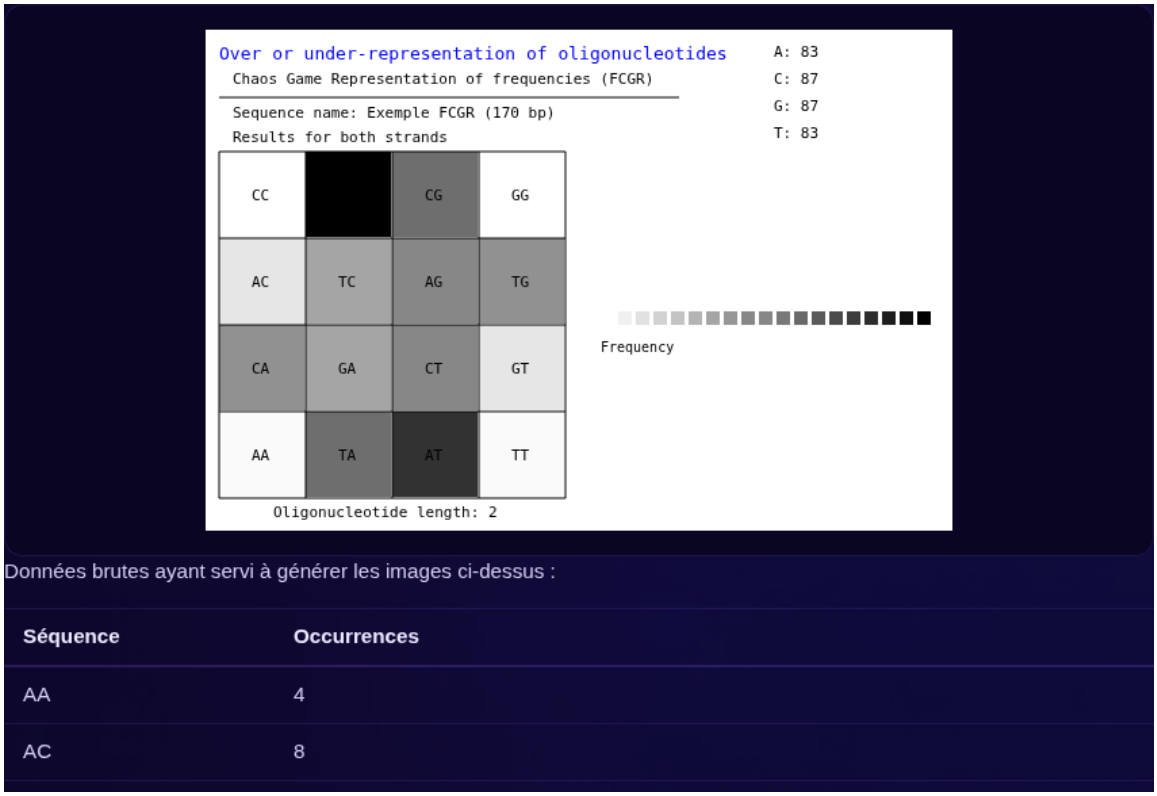

Figure 3.3: FCGR result : checkerboard of oligonucleotide frequencies, in grayscale.

Symfony implementation

```
public function cgrCompute(Request $request,
    ChaosGameRepresentationManager $manager, $form)
{
    $isComputed = false;
    $image = null;

    if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
```

```
        $formData = $form->getData();
        $image = basename(
            $manager->CGRCompute($formData["seq_name"], $formData["seq"
                ], $formData["size"])
        );
        $isComputed = true;
    }

    return $this->render('minitools/chaosGameRepresentationCGR.html.twig
        ', [
        'form'       => $form->createView(),
        'image'      => $image,
        'isComputed' => $isComputed,
    ]);
}
```

The template then displays the image returned by `CGRCompute()` with a simple `<img src="{{ path_graphs }}/{{ image }}">`.

Vanilla implementation

```
$manager = new ChaosGameRepresentationManager(
    $config['nucleotids_graphs'], // configuration array (see S3.2)
    new NucleotidApiAdapter($httpClient, $serializer)
);

$cheminSvg = $manager->CGRCompute('Ma sequence', $sequenceADN, 'auto');
// $cheminSvg points to the SVG file written to path_graphs
```

Warning

`CGRCompute()` and `createCGRImage()` write an SVG file to disk (see the configuration box in §3.2) : the `path_graphs` folder must exist and be writable. FCGR (§3.5) also writes its checkerboard to disk, in exactly the same way — its distinguishing feature isn't avoiding the file, but adding `<map>`/`<area>` HTML coordinates on top to make the image clickable.

3.4.2 Distance between sequences (oligonucleotides)

Compares several sequences against each other based on their oligonucleotide composition, and produces a dendrogram (relatedness tree) clustered using the UPGMA method. Two distance methods are offered : Pearson distance on tetranucleotide z-scores (suited to long sequences, over 20,000 bp), or Euclidean distance on the oligonucleotide frequencies of a chosen length.

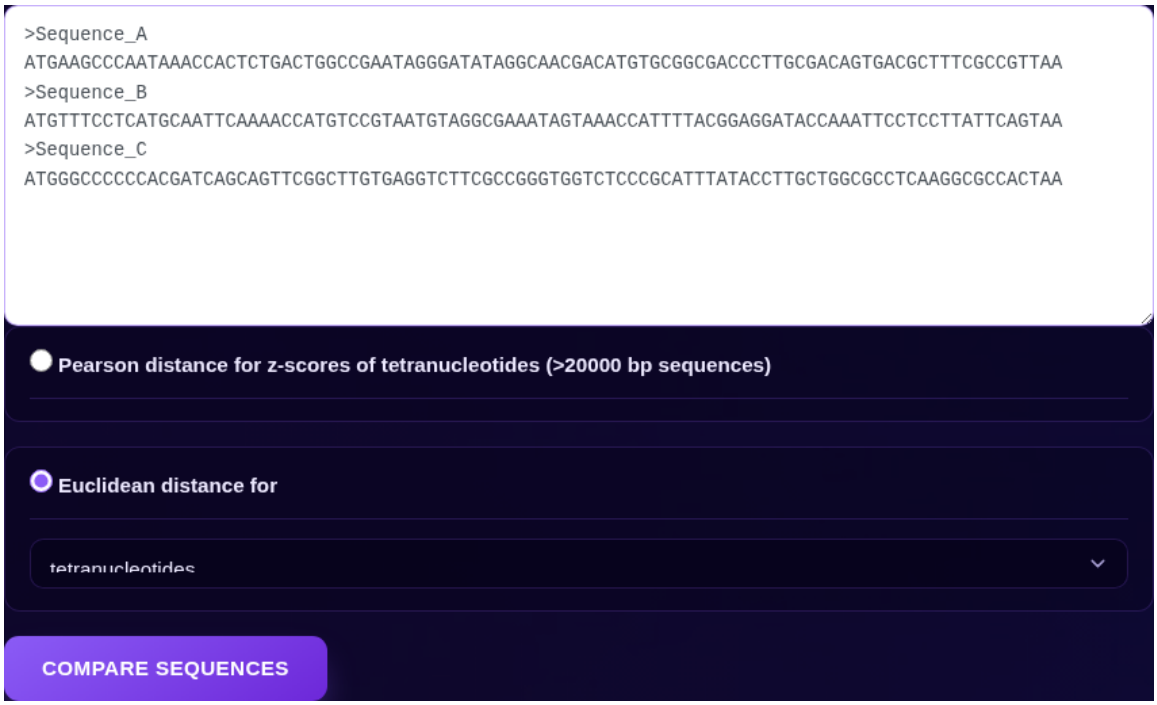

Figure 3.4: Sequence comparison form.

Field	Label / translation	Type	Values, default, role
`seq`	Sequence *Sequences*	long text	Multi-sequence FASTA format (`>name` followed by the sequence); 2,000,000 bp maximum in total.
`method`	Pearson distance… / Euclidean distance for *Distance method*	radio	`euclidean` (default) or `pearson`.
`len`	(oligonucleotide length) *Oligonucleotide length*	choice	di- to hexanucleotides (2 to 6); default : tetranucleotides (4). Ignored if method = Pearson.

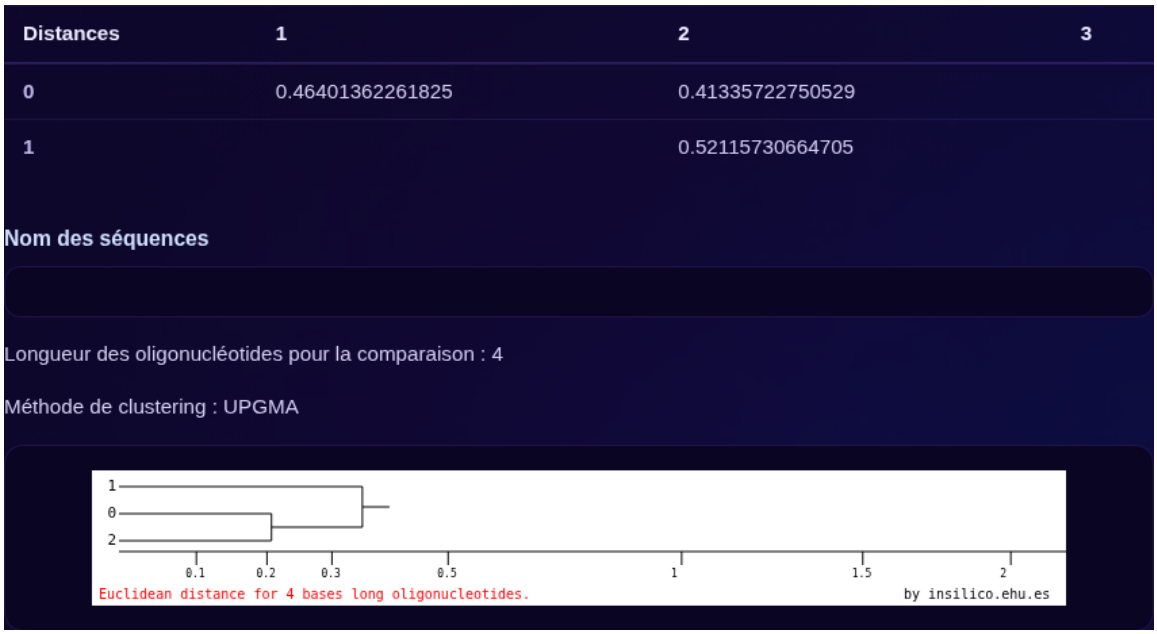

Figure 3.5: Distance matrix between sequences.

Symfony implementation

```php
public function distanceAmongSequencesAction(Request $request,
    DistanceAmongSequencesManager $manager)
{
    $form = $this->createForm(DistanceAmongSequencesType::class);
    $data = [];

    if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
        $formData = $form->getData();
        $length   = $formData["len"];
        $seqs     = $manager->formatSequences($formData["seq"]);

        if ($formData["method"] == "euclidean") {
            $oligoArray = $manager->
                computeOligonucleotidsFrequenciesEuclidean($seqs, $length
                );
            $data       = $manager->
                computeDistancesAmongFrequenciesEuclidean($seqs,
                $oligoArray, $length);
        } else {
            $oligoArray = $manager->computeOligonucleotidsFrequencies(
                $seqs);
            $data       = $manager->computeDistancesAmongFrequencies(
                $seqs, $oligoArray);
        }

        // the dendrogram is written as SVG alongside (see S3.5)
        $chemin = $graphsDir . '/dendogram_' . bin2hex(random_bytes(4))
            . '.svg';
        $manager->upgmaClustering($data, $formData["method"], $length,
            $chemin);
    }
```

```
24
25      return $this->render('minitools/distanceAmongSequences.html.twig',
26          ['form' => $form->createView(), 'data' => $data]);
27  }
```

Vanilla implementation

```
1  $manager = new DistanceAmongSequencesManager($oligosManager,
       $nucleotidApi);
2
3  $seqs = $manager->formatSequences($fastaMultiSequences);
4  $oligoArray = $manager->computeOligonucleotidsFrequenciesEuclidean($seqs
       , 4);
5  $distances = $manager->computeDistancesAmongFrequenciesEuclidean($seqs,
       $oligoArray, 4);
6
7  $manager->upgmaClustering($distances, 'euclidean', 4, '/path/to/
       dendrogram.svg');
```

Warning

Like CGR, `upgmaClustering()` writes the dendrogram to disk : the same requirement on `path_graphs` applies (existing, writable, served publicly).

3.4.3 DNA to protein translation

Translates a DNA sequence into protein, in one or more reading frames, using either a predefined genetic code (compiled by Elzanowski and Ostell, NCBI) or a custom one, and optionally searches for open reading frames (ORFs).

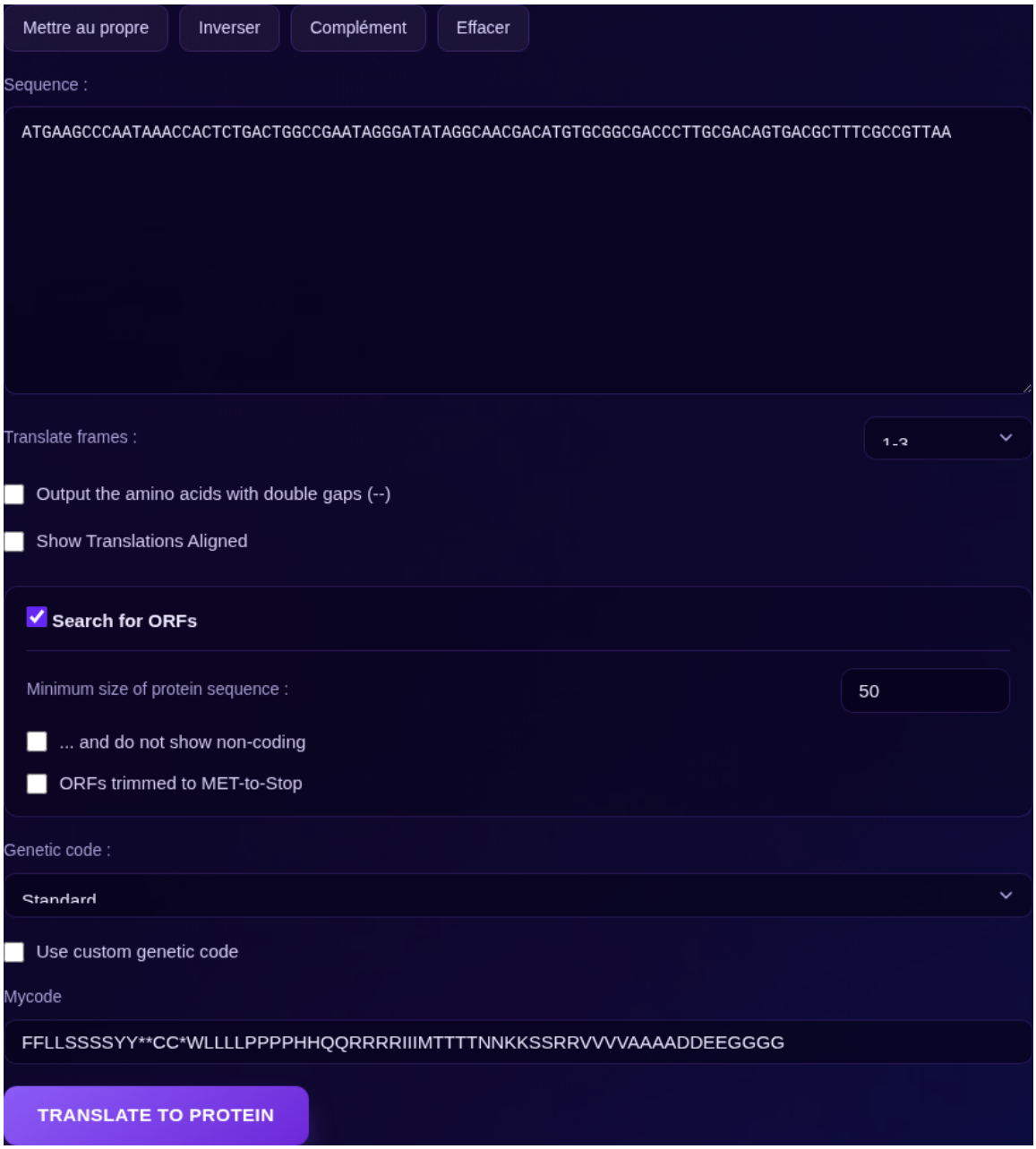

Figure 3.6: DNA to protein translation form.

Field	Label / translation	Type	Values, default, role
`sequence`	Sequence *Sequence*	long text	DNA to be translated.
`frames`	(reading frames) *Reading frames*	choice	E.g. : frame 1, 2, 3, or all three.
`genetic_code`	Genetic code *Genetic code*	choice	Standard by default; list of NCBI codes.
`usemycode`	Use my own code *Use my own code*	checkbox	Enables the `mycode` field.
`mycode`	Custom genetic code *Custom genetic code*	text	A 64-character string (one per codon); an example (standard code) is pre-filled.

Field	Label / translation	Type	Values, default, role
`search_orfs`	Search ORFs *Search ORFs*	checkbox	
`protsize`	Minimum protein size *Minimum protein size*	text	50 by default; only used when searching for ORFs.
`only_coding`	Only ORFs starting with Met *Only ORFs starting with Methionine*	checkbox	
`trimmed`	Trim ORFs from Met to Stop *Trim from Methionine to Stop*	checkbox	
`show_aligned`	Show translations aligned *Show translations aligned*	checkbox	
`dgaps`	Show gaps as dashes *Show gaps as dashes*	checkbox	

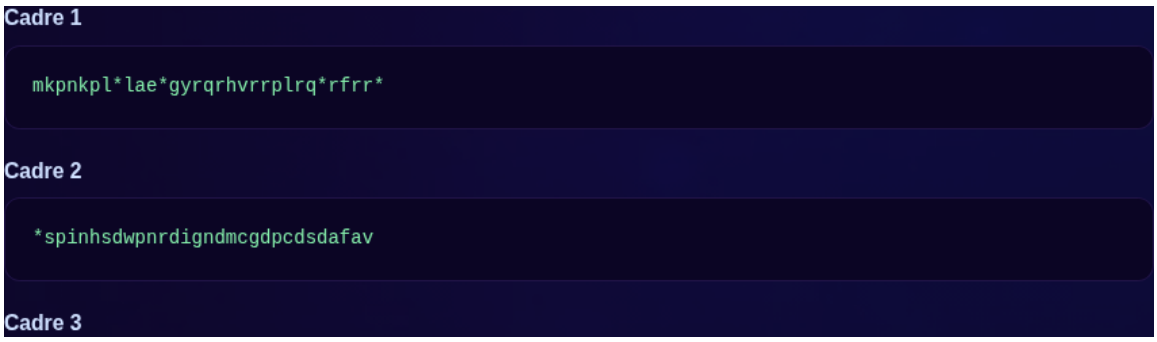

Figure 3.7: Translations obtained, one block per reading frame.

Symfony implementation

```
public function dnaToProteinAction(Request $request, DnaToProteinManager
    $manager)
{
    $form = $this->createForm(DnaToProteinType::class);
    $frames = [];

    if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
        $formData = $form->getData();
        $sequence = $formData["sequence"];

        if ($formData["usemycode"]) {
            $frames = $manager->customTreatment($formData["frames"],
                $sequence, $formData["mycode"]);
        } else {
            $frames = $manager->definedTreatment($formData["frames"],
                $formData["genetic_code"], $sequence);
```

```
14          }
15
16          if ($formData["search_orfs"]) {
17              $frames = $manager->findORF($frames, $formData["protsize"],
18                  (bool) $formData["only_coding"], (bool) $formData["
                        trimmed"]);
19          }
20      }
21
22      return $this->render('minitools/dnaToProtein.html.twig',
23          ['form' => $form->createView(), 'frames' => $frames]);
24  }
```

The template can then rely on the Twig extension (§3.3) rather than precomputing the alignment in the controller : `{{ show_translations_aligned(sequence, frame) }}`.

Vanilla implementation

```
1  $manager = new DnaToProteinManager($aminoApi, $tripletApi,
       $tripletSpecieApi);
2
3  $frames = $manager->definedTreatment(1, 'Standard', $sequenceADN);
4  $frames = $manager->findORF($frames, 50, true, true);
```

3.4.4 Palindromic sequence search

A DNA sequence whose 5'→3' reading is identical on both strands — in other words, a sequence equal to its own reverse complement. This is the case for the recognition sites of many restriction enzymes.

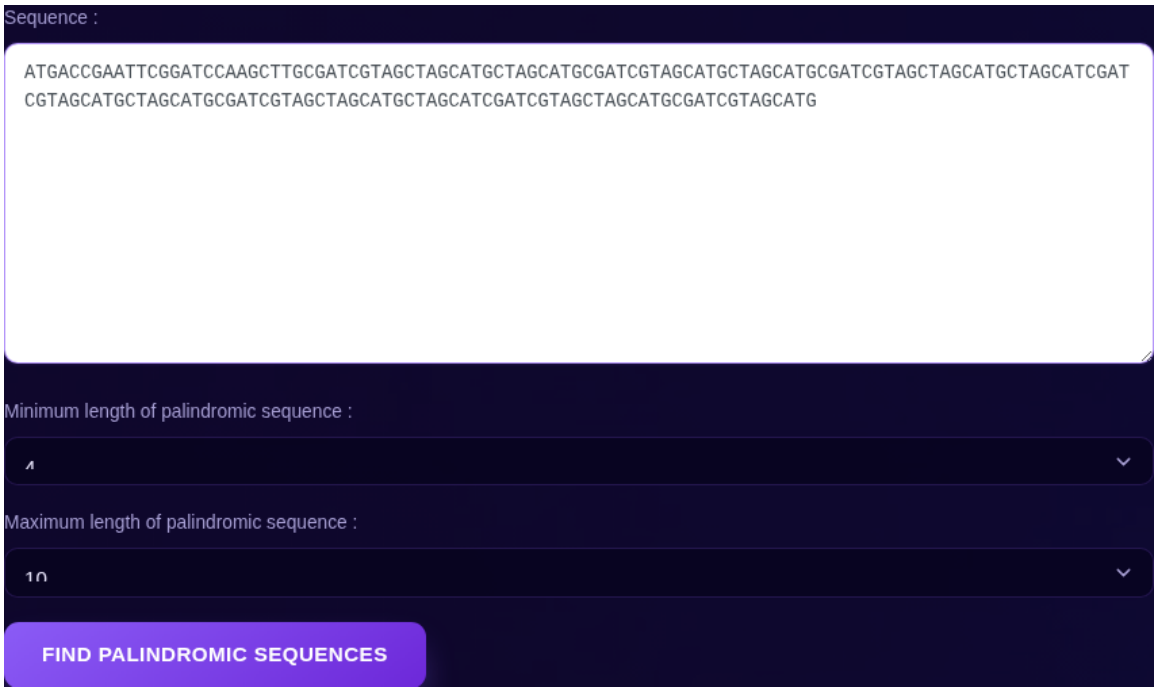

Figure 3.8: Palindrome search form.

Field	Label / translation	Type	Values, default, role
`seq`	Sequence *Sequence*	long text	DNA to analyze.
`min`	Minimum length *Minimum length*	choice	Minimum length of a sought-after palindrome.
`max`	Maximum length *Maximum length*	choice	Maximum length of a sought-after palindrome.

Position	Séquence
4	CCGAATTCGG
5	CGAATTCG
6	GAATTC
7	AATT
12	GGATCC
13	GATC
17	CAAGCTTG
18	AAGCTT
19	AGCT
25	CGATCG
26	GATC
31	TAGCTA
32	AGCT
33	GCTAGC
34	CTAG
35	TAGCATGCTA

Figure 3.9: Positions and palindromic sequences found.

Symfony implementation

```
public function findPalindromesAction(Request $request,
    FindPalindromeManager $manager)
{
     $palindromes = [];
     $form = $this->createForm(FindPalindromesType::class);

     if ($request->isMethod('POST') && $form->handleRequest($request)->
         isValid()) {
          $formData = $form->getData();
          $palindromes = $manager->findPalindromicSeqs(
```

```
            $formData["seq"], $formData["min"], $formData["max"]
        );
    }

    return $this->render('minitools/findPalindromes.html.twig',
        ['form' => $form->createView(), 'palindromes' => $palindromes]);
}
```

Vanilla implementation

```
$manager = new FindPalindromeManager();
$palindromes = $manager->findPalindromicSeqs($sequenceADN, 4, 10);
// ['position_in_the_sequence' => 'palindromic_sub-sequence', ...]
```

</content> </invoke>

Note

The only tool in the chapter that does not depend on any API adapter : `FindPalindromeManager` only uses the `SequenceTrait` from BioPHP (reverse complement computed locally). This is the simplest way to test the "vanilla " implementation described here.

3.4.5 GC content finder

Computes the length and base composition (A, C, G, T) of a sequence supplied as an uploaded FASTA file.

Figure 3.10: FASTA file upload form.

Field	Label / translation	Type	Values, default, role
`fasta`	FASTA File *FASTA file*	file	100,000 KB maximum; no MIME type check (a `.fasta`/`.fa` file sent by the browser as `application/octet-stream` is accepted).

Figure 3.11: Length and base composition of the uploaded file.

Symfony implementation

```php
public function fastaUploaderAction(Request $request,
    FastaUploaderManager $manager)
{
    $length = 0; $a = 0; $g = 0; $t = 0; $c = 0;
    $form = $this->createForm(FastaUploaderType::class);

    if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
        $formData = $form->getData();

        $var = $manager->createFiles($formData["fasta"], $this->
            getParameter('brochures_directory'));

        // createFiles() returns the RAW content of the file: the ">..."
            header and the
        // newlines must be stripped before checkNucleotidSequence(),
            otherwise a real
        // .fasta upload gets rejected as "invalid"
        $sequence = preg_replace('/^>.*$/m', '', $var);
        $sequence = preg_replace('/\s+/', '', $sequence);

        $manager->checkNucleotidSequence($sequence, $a, $g, $t, $c,
            strlen($sequence));
        $length = strlen($sequence);
    }

    return $this->render('minitools/gcContentFinder.html.twig',
        ['form' => $form->createView(), 'length' => $length, 'a' => $a,
            'g' => $g, 't' => $t, 'c' => $c]);
}
```

Vanilla implementation

```php
$manager = new FastaUploaderManager();

$contenuBrut = $manager->createFiles($fichierUploade, '/chemin/vers/
    uploads');
$sequence = preg_replace('/^>.*$/m', '', $contenuBrut);
$sequence = preg_replace('/\s+/', '', $sequence);

$manager->checkNucleotidSequence($sequence, $a, $g, $t, $c, strlen(
    $sequence));
```

Warning

`createFiles()` writes the uploaded file to disk, then reads it back to extract its content : the same requirement for a writable folder applies as for the tools that generate an SVG. This is also one of only two tools in the chapter (along with distance between sequences) that go through a disk write.

Note

`checkNucleotidSequence()` rejects any character that is not strictly A/T/G/C (not even degenerate nucleotides). Removing the `>` header line and the newlines before the call, as shown above, is essential for a real FASTA file to pass validation.

3.4.6 Melting temperature (Tm)

Estimates the melting temperature (Tm) of a DNA duplex from a primer sequence, using either of two possible approximations : the basic formula (Wallace, based on primer length) or the nearest-neighbor stacking calculation (SantaLucia 1998; von Ahsen *et al.* 1999).

Note

For sequences shorter than 14 nucleotides : $T_m = (w_A + x_T) \times 2 + (y_G + z_C) \times 4$. Above that : $T_m = 64.9 + 41 \times (y_G + z_C - 16.4)/(w_A + x_T + y_G + z_C)$. Both formulas assume hybridization at 50 nM primer, 50 mM Na^+ and pH 7.0. Degenerate nucleotides (Y, R, W, S, K, M, D, V, H, B, N) are substituted internally before the calculation, once for the minimum Tm and once for the maximum Tm.

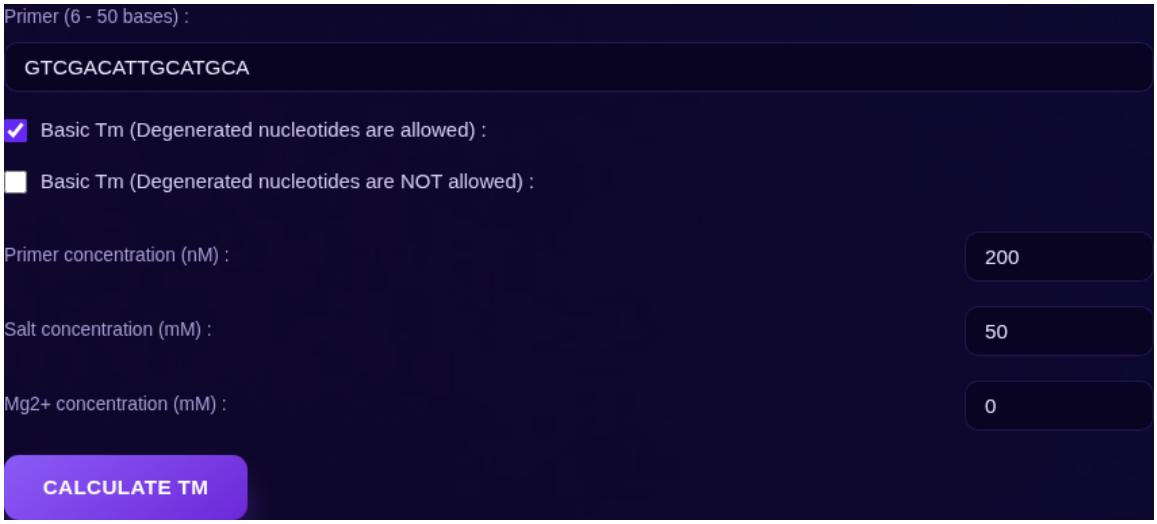

Figure 3.12: Melting temperature calculation form.

Field	Label / translation	Type	Values, default, role
`primer`	Primer (6 - 50 bases) *Primer (6 to 50 bases)*	text	Validated by the `MeltingTemperature` constraint (NC-IUBMB composition, length 6-50 bp).
`basic`	Basic Tm (Degenerated nucleotides are allowed) *Basic Tm (degenerate nucleotides allowed)*	checkbox	

Field	Label / translation	Type	Values, default, role
`nearestNeighbor`	Basic Tm (Degenerated nucleotides are NOT allowed) *Nearest-neighbor stacking Tm (degenerate nucleotides excluded)*	checkbox	
`cp`	Primer concentration (nM) *Primer concentration (nM)*	text	200 by default.
`cs`	Salt concentration (mM) *Salt concentration (mM)*	text	50 by default.
`cmg`	Mg2+ concentration (mM) *Mg^{2+} concentration (mM)*	text	0 by default.

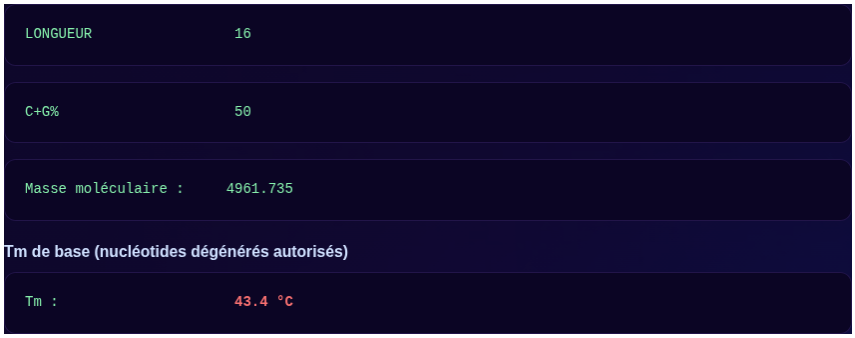

Figure 3.13: Length, %GC, molecular weight and Tm computed.

Symfony implementation

```
public function meltingTemperatureAction(Request $request,
    MeltingTemperatureManager $manager)
{
    $cg = $upperMwt = $lowerMwt = $countATGC = $tmMin = $tmMax = 0;
    $aTmBaseStacking = [];
    $form = $this->createForm(MeltingTemperatureType::class);

    if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
        $formData = $form->getData();
        $primer   = $formData["primer"];

        $cg = $manager->calculateCG($primer);
        $manager->calculateMWT($upperMwt, $lowerMwt, $primer);
        $manager->basicCalculations($formData["basic"], $primer,
            $countATGC, $tmMin, $tmMax);
        $manager->neighborCalculations(
```

```
            $formData["nearestNeighbor"], $aTmBaseStacking, $primer,
            $formData["cp"], $formData["cs"], $formData["cmg"]
        );
    }

    return $this->render('minitools/meltingTemperature.html.twig',
        ['form' => $form->createView(), 'tmMin' => $tmMin, 'tmMax' =>
            $tmMax]);
}
```

Vanilla implementation

```
$manager = new MeltingTemperatureManager($sequenceManager,
    $tmBaseStackingApi);

$cg = $manager->calculateCG('GTCGACATTGCATGCA');
$manager->basicCalculations(true, 'GTCGACATTGCATGCA', $countATGC, $tmMin
    , $tmMax);
// $tmMin / $tmMax hold the computed Tm
```

3.4.7 Microarray data analysis

An adaptive quantification method with local background subtraction : for each spot, the value (signal − background) is expressed as a percentage of the total sum, across both channels; the ch1/ch2 and ch2/ch1 ratios of the replicates of a given gene are then summarized by their median and its base-10 logarithm. A color code flags over-expression (red) or under-expression (blue) in the demo's rendering.

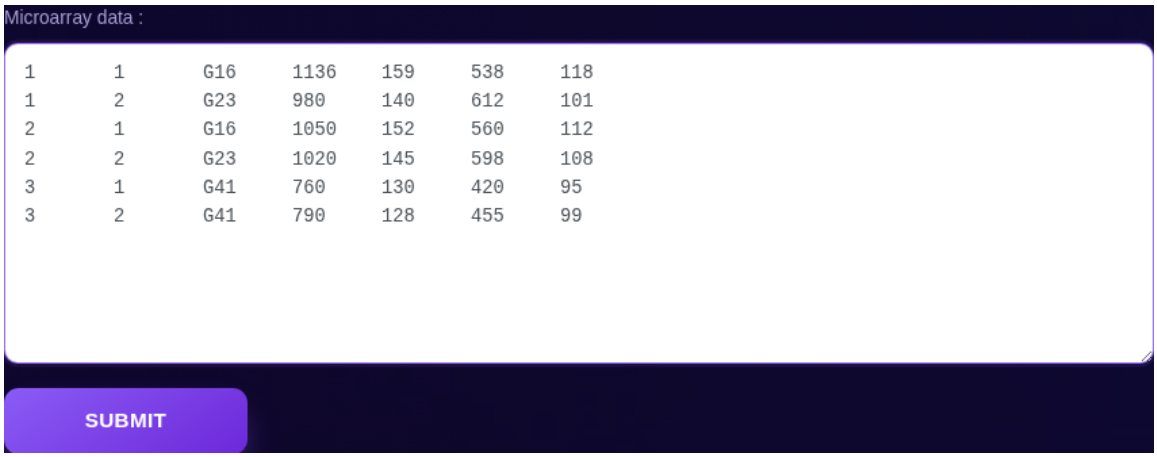

Figure 3.14: Microarray data upload form.

Field	Label / translation	Type	Values, default, role
`data`	Microarray data *Microarray data*	long text	Raw tab-separated table : column, row, gene name, signal and background of both channels. A sample dataset (10 genes) is pre-filled.

Identification	Nb de données	Canal 1	log10	Canal 2	log10
G16	2	1.131	0.053	0.889	-0.051
G23	2	0.896	-0.048	1.118	0.049
G41	2	0.992	-0.004	1.009	0.004

Figure 3.15: Count, medians and logarithms per gene.

Symfony implementation

```php
public function microArrayAnalysisAdaptiveQuantificationAction(Request
    $request,
     MicroarrayAnalysisAdaptiveManager $manager)
{
     $results = [];
     $form = $this->createForm(MicroArrayDataAnalysisType::class);

     if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
         $formData = $form->getData();
         $results = $manager->
            processMicroarrayDataAdaptiveQuantificationMethod($formData["
            data"]);
     }

     return $this->render('minitools/
        microArrayAnalysisAdaptiveQuantification.html.twig',
         ['form' => $form->createView(), 'results' => $results]);
}
```

Vanilla implementation

```php
$manager = new MicroarrayAnalysisAdaptiveManager();
$resultats = $manager->processMicroarrayDataAdaptiveQuantificationMethod
    ($donneesBrutes);
// $resultats['GeneName'] = ['n_data' => .., 'median1' => .., 'medlog1'
    => .., ...]
```

Note

The expected format is strict : each line must have at least 7 tab-separated columns (position, position, name, channel 1 signal, channel 1 background, channel 2 signal, channel 2 background) — shorter lines are silently ignored. A signal lower than or equal to the background (a weak or failed spot) is clamped to a small positive value instead of producing an undefined logarithm.

3.4.8 Microsatellite repeat search

Searches for simple tandem repeats (di-, tri-, tetra-, pentanucleotide…) in a sequence — also known as *Simple Tandem Repeats* (STR) or *Simple Sequence Repeats* (SSR), generally shorter than 100 bases.

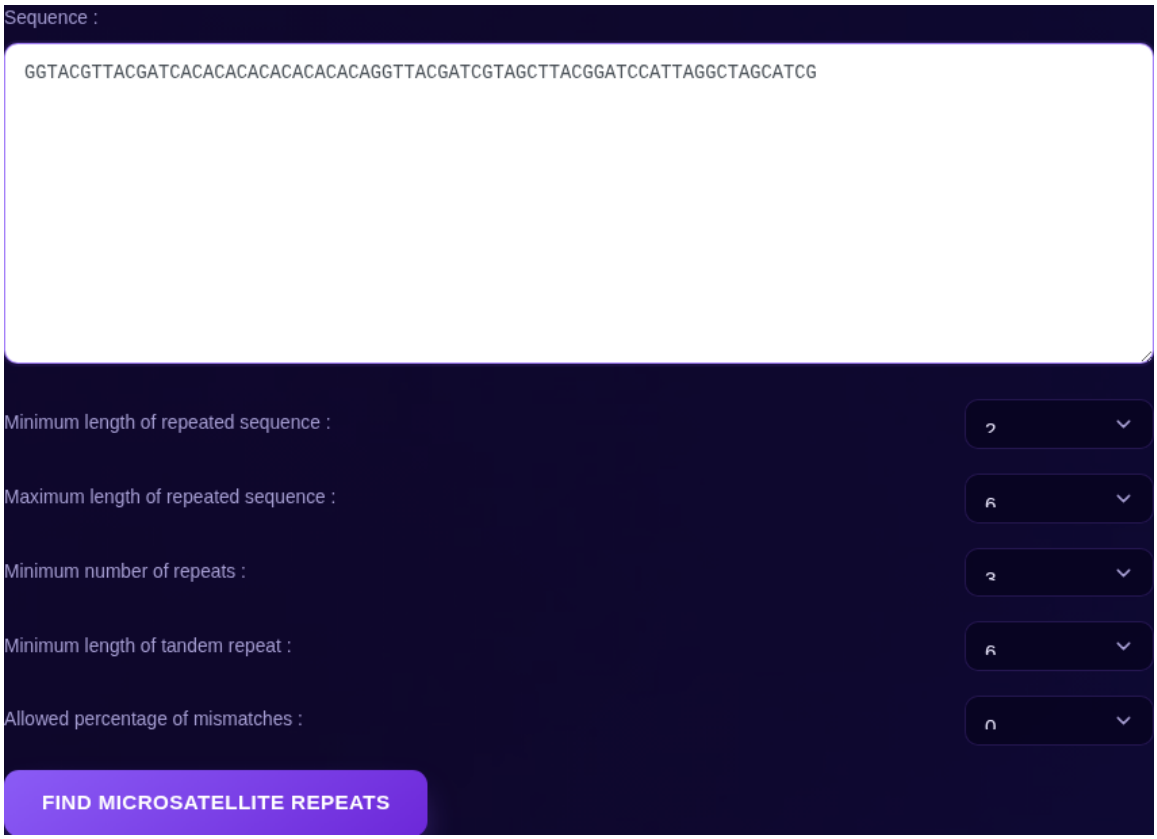

Figure 3.16: Microsatellite search form.

Field	Label / translation	Type	Values, default, role
`sequence`	Sequence *Sequence*	long text	DNA to analyze; an example is pre-filled.
`min`	Minimum length of repeated sequence *Minimum length of the repeated motif*	choice	2 to 6.
`max`	Maximum length of repeated sequence *Maximum length of the repeated motif*	choice	3 to 10; default : 6.
`min_repeats`	Minimum number of repeats *Minimum number of repeats*	choice	2 to 6; default : 3.
`length_of_MR`	Minimum length of tandem repeat *Minimum length of the tandem repeat*	choice	5 to 20; default : 6.
`mismatch`	Allowed percentage of mismatches *Allowed percentage of mismatches*	choice	0, 10, 20 or 30 %.

Position	Cycle	Répétitions	Séquence
13	2	9	CACACACACACACACACA

Figure 3.17: Position, period, number of repeats and motif found.

Symfony implementation

```
public function microsatelliteRepeatsFinderAction(Request $request,
    MicrosatelliteRepeatsFinderManager $manager)
{
    $results = [];
    $form = $this->createForm(MicrosatelliteRepeatsFinderType::class);

    if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
        $formData = $form->getData();
        $results = $manager->findMicrosatelliteRepeats(
            $formData["sequence"], $formData["min"], $formData["max"],
            $formData["min_repeats"], $formData["length_of_MR"],
                $formData["mismatch"]
        );
    }

    return $this->render('minitools/microsatelliteRepeatsFinder.html.
        twig',
        ['form' => $form->createView(), 'results' => $results]);
}
```

Vanilla implementation

```
$manager = new MicrosatelliteRepeatsFinderManager();
$resultats = $manager->findMicrosatelliteRepeats($sequenceADN, 2, 6, 3,
    6, 0);
```

3.4.9 (Oligo)nucleotide frequency

Counts the frequency of each oligonucleotide of a given length (1 to 8 bases) in a sequence, on a single strand or on both.

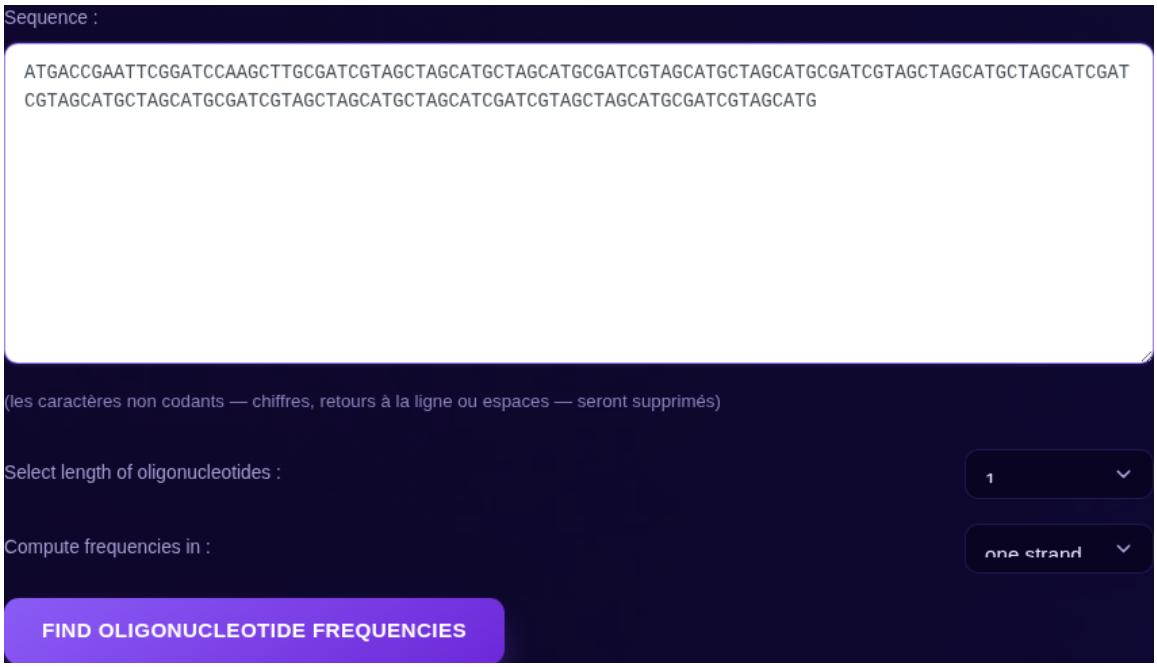

Figure 3.18: Oligonucleotide frequency calculation form.

Field	Label / translation	Type	Values, default, role
`sequence`	Sequence *Sequence*	long text	DNA to analyze; an example is pre-filled. Must be at least 4^{len} bases long.
`len`	Select length of oligonucleotides *Oligonucleotide length*	choice	1 to 8.
`strands`	Compute frequencies in *Compute frequencies on*	choice	one strand or both.

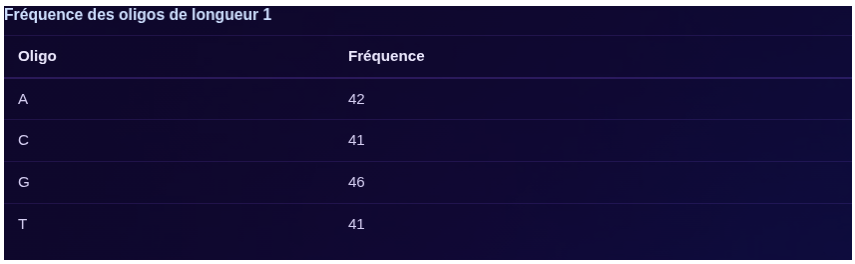

Figure 3.19: Frequency of each oligonucleotide found.

Symfony implementation

```
public function oligonucleotideFrequencyAction(Request $request,
    NucleotidApiAdapter $nucleotidApi, OligosInterface $oligos)
{
    $results = []; $length = 0;
```

```
    $form = $this->createForm(OligoNucleotideFrequencyType::class);

    if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
        $formData = $form->getData();
        $length   = $formData["len"];
        $sequence = $formData["sequence"];

        if ($formData["strands"] == 2) {
            $complements = $nucleotidApi::GetDNAComplement($nucleotidApi
                ->getNucleotids());
            GeneticsFunctions::createInversion($sequence, $complements);
        }

        $results = $oligos->findOligos($sequence, $length);
        ksort($results);
    }

    return $this->render('minitools/oligonucleotideFrequency.html.twig',
        ['form' => $form->createView(), 'results' => $results]);
}
```

Vanilla implementation

```
$oligos = new OligosManager(); // from BioPHP, see Chapter 2
$resultats = $oligos->findOligos($sequenceADN, 1);
// $resultats['A'] = 42, $resultats['C'] = 41, ...
```

Note

The only tool in the chapter without its own manager : it directly reuses BioPHP's `OligosManager` (Chapter 2, §2.7). To compute on both strands, the controller itself builds the reverse complement strand before the call — `findOligos()` does not handle this on its own.

3.4.10 *In silico* PCR amplification

Simulates a PCR amplification : searches a sequence for all positions where primer 1 (forward) is followed, within the specified maximum length, by the reverse complement of primer 2, and lists the amplicons found with their position and length.

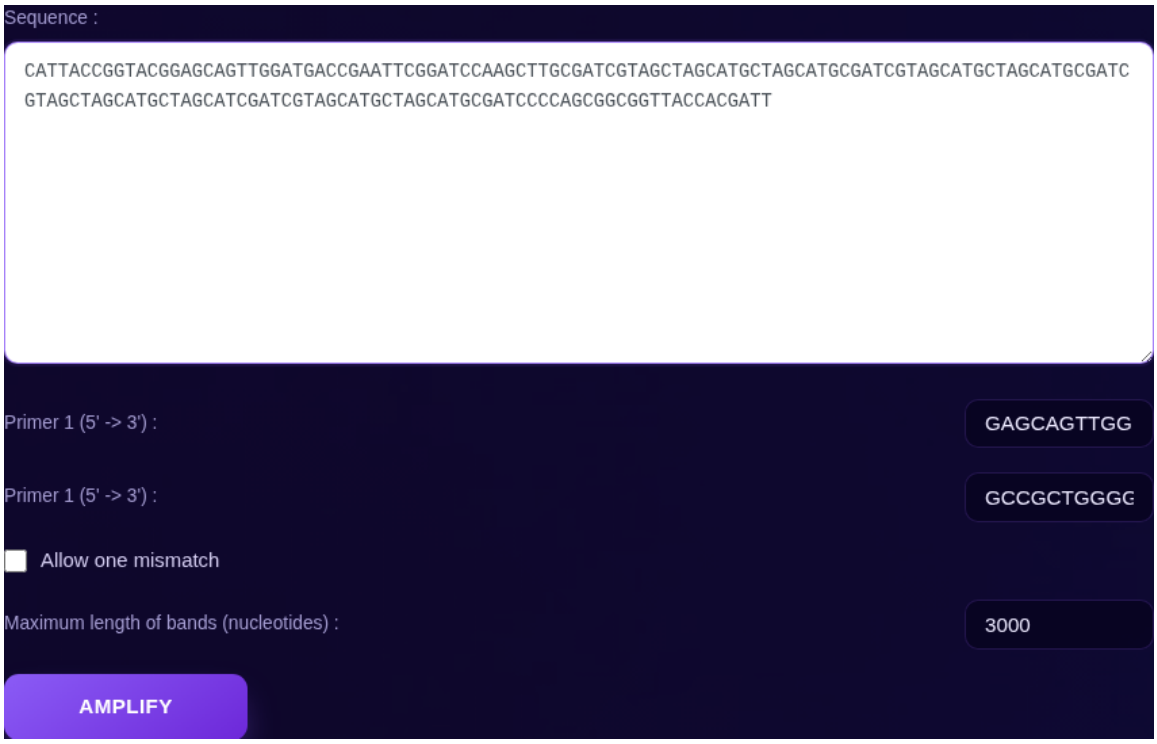

Figure 3.20: PCR amplification form.

Field	Label / translation	Type	Values, default, role
`sequence`	Sequence *Sequence*	long text	DNA in which to search for the amplicon; an example is pre-filled.
`primer1`	Primer 1 (5' -> 3') *Primer 1 (5' to 3')*	text	`GAGCAGTTGG` by default.
`primer2`	Primer 2 (5' -> 3') *Primer 2 (5' to 3')*	text	`GCCGCTGGGG` by default.
`allowmismatch`	Allow one mismatch *Allow one mismatch*	checkbox	
`length`	Maximum length of bands (nucleotides) *Maximum band length (nucleotides)*	text	3,000 by default.

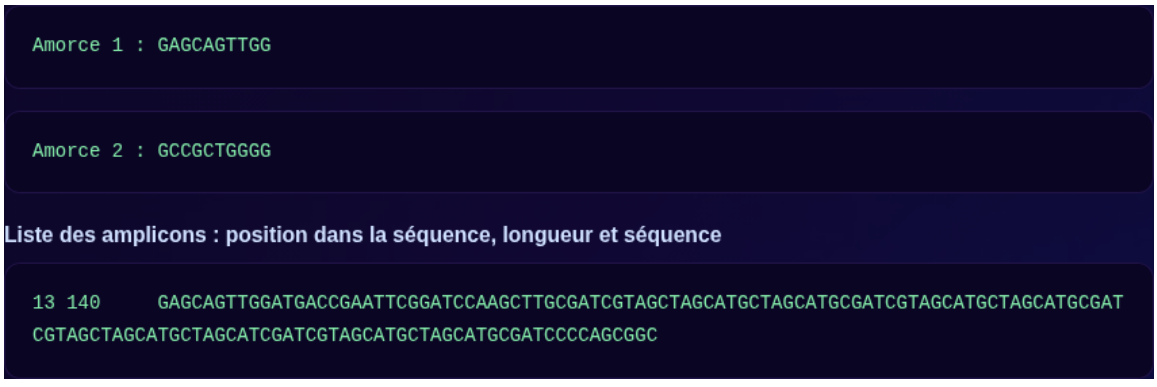

Figure 3.21: Amplicons found : position, length and sequence.

Symfony implementation

```php
public function pcrAmplificationAction(Request $request,
    PcrAmplificationManager $manager)
{
    $results = [];
    $form = $this->createForm(PcrAmplificationType::class);

    if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
        $formData = $form->getData();

        $startPattern = $manager->createStartPattern(
            $formData["primer1"], $formData["primer2"], $formData["
                allowmismatch"]
        );
        $endPattern = $manager->createEndPattern($startPattern);

        $results = $manager->amplify($startPattern, $endPattern,
            $formData["sequence"], $formData["length"]);
    }

    return $this->render('minitools/pcrAmplification.html.twig',
        ['form' => $form->createView(), 'results' => $results]);
}
```

Vanilla implementation

```php
$manager = new PcrAmplificationManager($nucleotidApi);

$startPattern = $manager->createStartPattern('GAGCAGTTGG', 'GCCGCTGGGG',
    false);
$endPattern   = $manager->createEndPattern($startPattern);
$amplicons    = $manager->amplify($startPattern, $endPattern,
    $sequenceADN, 3000);
// $amplicons[starting_position] = amplicon_length
```

Warning

The search scheme is counter-intuitive at first : `createEndPattern()` does not work on primer 2 alone, but on the reverse complement of the entire `startPattern` (the two primers separated by a regular "or "). In practice, this amounts to searching for primer 1 as-is at the start of the amplicon, then the reverse complement of primer 2 at the end of the amplicon — exactly what is expected in a real PCR, where primer 2 hybridizes on the complementary strand.

3.4.11 Protein sequence properties

Computes, on all or part of a protein sequence : amino acid composition, molecular weight, molar absorption coefficient, isoelectric point, charge at a given pH, and displays the sequence in 3-letter code or colored by the physicochemical nature of the residues. NC-IUBMB codes serve as the reference, and non-coding characters are removed by default.

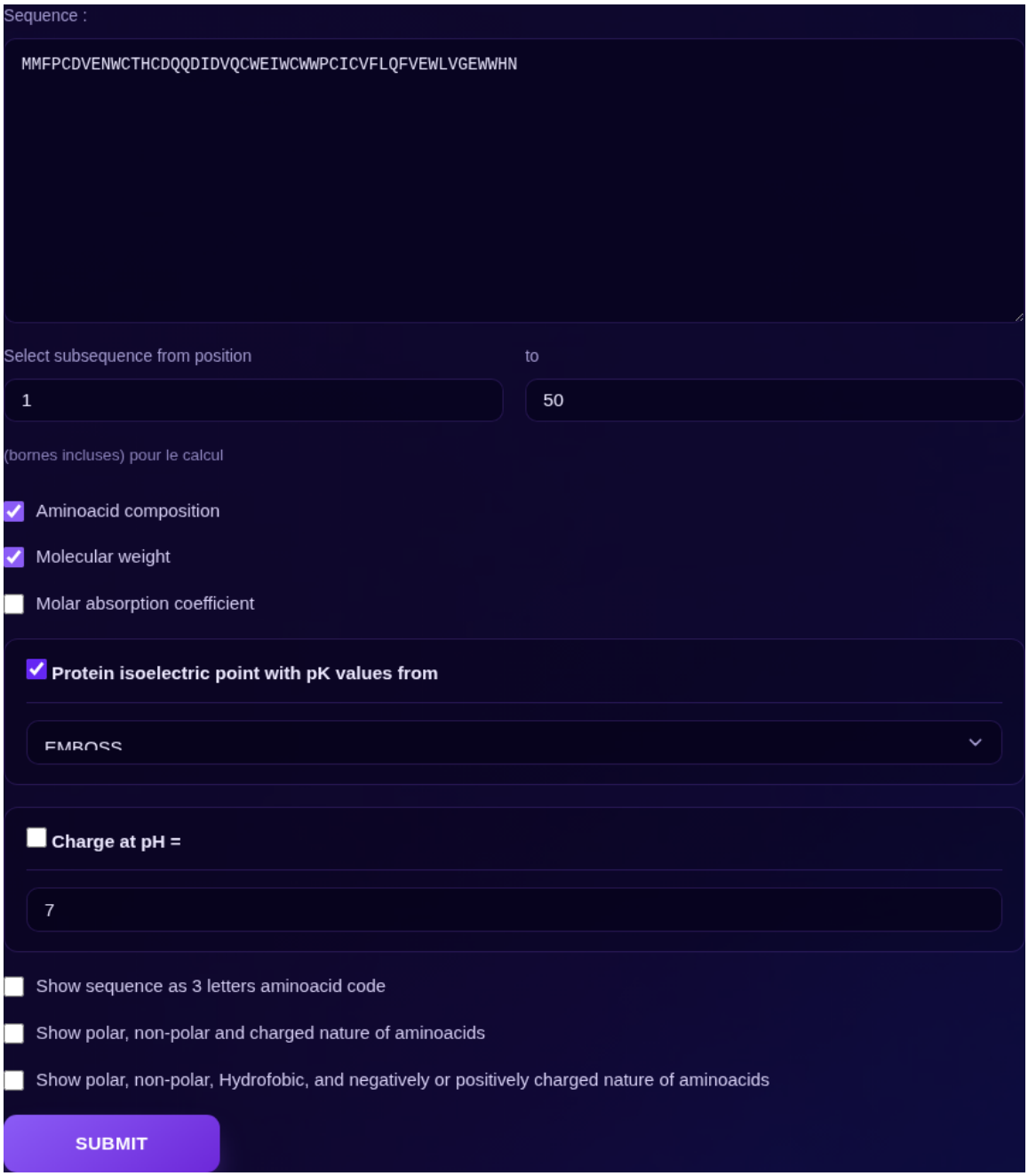

Figure 3.22: Protein sequence property calculation form.

Field	Label / translation	Type	Values, default, role
`seq`	Sequence *Sequence*	long text	Protein to analyze.
`start`	Select subsequence from position *Subsequence starting at position*	text	Empty by default (see the warning box below).
`end`	to *to position*	text	Empty by default.
`composition`	Aminoacid composition *Amino acid composition*	checkbox	

Field	Label / translation	Type	Values, default, role
`molweight`	Molecular weight *Molecular weight*	checkbox	
`abscoef`	Molar absorption coefficient *Molar absorption coefficient*	checkbox	
`charge`	Protein isoelectric point with pK values from *Isoelectric point, pK values from*	checkbox	Uses the reference set chosen in `data_source`.
`data_source`	(pK reference set) *pK reference set*	choice	EMBOSS / DTASelect / Solomon.
`charge2`	Charge at pH = *Charge at pH*	checkbox	
`pH`	(pH value) *pH*	text	7 by default.
`three_letters`	Show sequence as 3 letters aminoacid code *Show as 3-letter code*	checkbox	
`type1`	Show polar, non-polar and charged nature *Polar / non-polar / charged nature*	checkbox	Polar : SCTNQHYW; non-polar : GAPVILFM; charged : DEKR.
`type2`	Show polar, non-polar, hydrophobic... nature *Detailed nature (5 categories)*	checkbox	

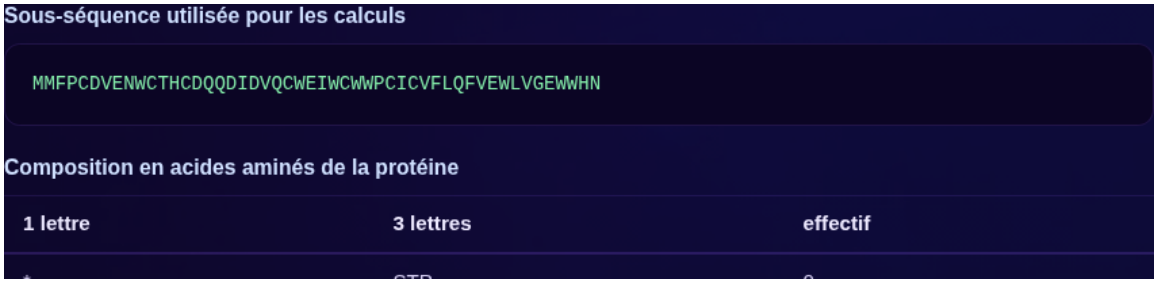

Figure 3.23: Composition, molecular weight and other properties computed.

Symfony implementation

```
public function proteinPropertiesAction(Request $request,
    ProteinPropertiesManager $manager)
{
     $aminoacids = [];
     $molweight = 0;
     $form = $this->createForm(ProteinPropertiesType::class);

     if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
```

```
        $formData = $form->getData();

        $seq = GeneticsFunctions::removeNonCodingProt($formData["seq"]);
        $subsequence = $manager->writeSubsequence($formData["start"],
            $formData["end"], $seq);
        $aminoacidContent = $manager->aminoacidContent($subsequence);

        $manager->setPk($formData["data_source"]);

        if ($formData["composition"]) {
            $aminoacids = $manager->formatAminoacidContent(
                $aminoacidContent);
        }
        if ($formData["molweight"]) {
            $molweight = $manager->proteinMolecularWeight(
                $aminoacidContent);
        }
    }

    return $this->render('minitools/proteinProperties.html.twig',
        ['form' => $form->createView(), 'aminoacids' => $aminoacids, '
            molweight' => $molweight]);
}
```

Vanilla implementation

```
$manager = new ProteinPropertiesManager($pkApi);

$sous_sequence = $manager->writeSubsequence(1, strlen($proteine),
    $proteine);
$composition = $manager->aminoacidContent($sous_sequence);
$manager->setPk('EMBOSS');
$masse = $manager->proteinMolecularWeight($composition);
```

Warning

`writeSubsequence($start, $end, $seq)` only returns the whole sequence if **at least one** of the two fields is filled in ; if `start` **and** `end` are both empty, the returned subsequence is an **empty string**, and every calculation that depends on it (composition, weight, etc.) silently comes out at zero without raising any error. Always fill in at least `start=1` and `end=` the length of the sequence to analyze it in full.

3.4.12 Protein to DNA back-translation

Back-translates a protein sequence into a DNA sequence, choosing at each position the most likely codon for the chosen species (or genetic code) — the genetic codes and their codons come from BioAPI (Chapter 1).

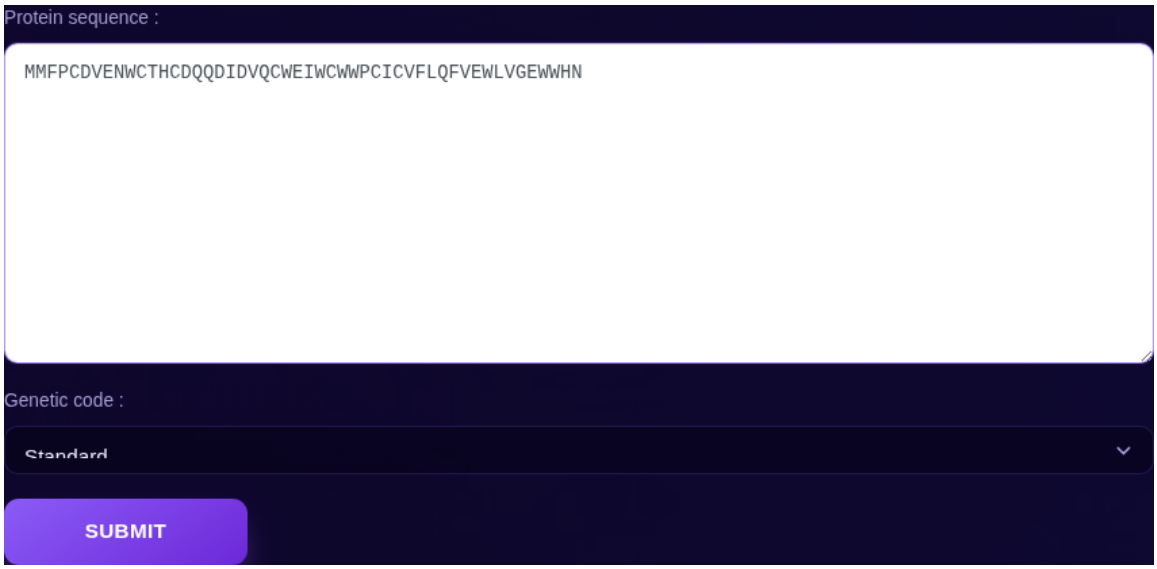

Figure 3.24: Protein to DNA back-translation form.

Field	Label / translation	Type	Values, default, role
`sequence`	Protein sequence *Protein sequence*	long text	`FLIMVSPTAYHQNKDECWRGX*` by default (all valid 1-letter codes).
`genetic_code`	Genetic code *Genetic code*	choice	List of species/codes available on BioAPI, fetched dynamically when the form loads.

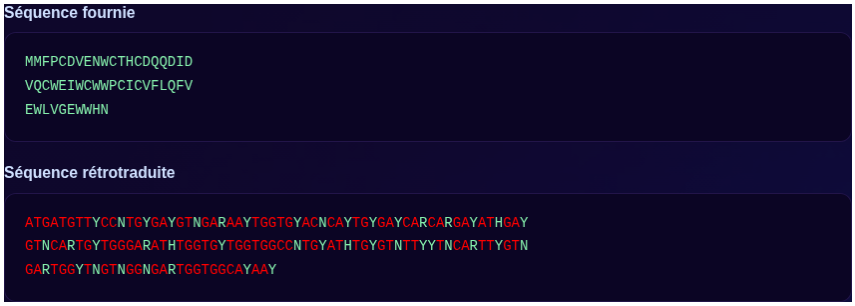

Figure 3.25: Back-translated DNA sequence.

Symfony implementation

```
1  public function proteinToDnaAction(Request $request, ProteinToDnaManager
       $manager)
2  {
3      $dna = "";
4      $form = $this->createForm(ProteinToDnaType::class);
5
6      if ($request->isMethod('POST') && $form->handleRequest($request)->
           isValid()) {
7          $formData = $form->getData();
```

```
8           $dna = $manager->translateProteinToDNA($formData["sequence"],
                $formData["genetic_code"]);
9       }
10
11      return $this->render('minitools/proteinToDna.html.twig',
12          ['form' => $form->createView(), 'dna' => $dna]);
13  }
```

Vanilla implementation

```
1  $manager = new ProteinToDnaManager($tripletSpecieApi);
2  $adn = $manager->translateProteinToDNA('MAVLK', 'Standard');
```

Note

`ProteinToDnaType` queries `TripletSpecieApiAdapter` as soon as it is built, to populate the list of genetic codes : in "vanilla ", you have to make this network call yourself before showing the available choices to the user.

3.4.13 Random sequences

Generates a random sequence using one of three methods : shuffling the characters of an existing sequence, drawing DNA nucleotides according to chosen A/C/G/T frequencies, or drawing amino acids according to 20 chosen frequencies (the default values are those of the human genome).

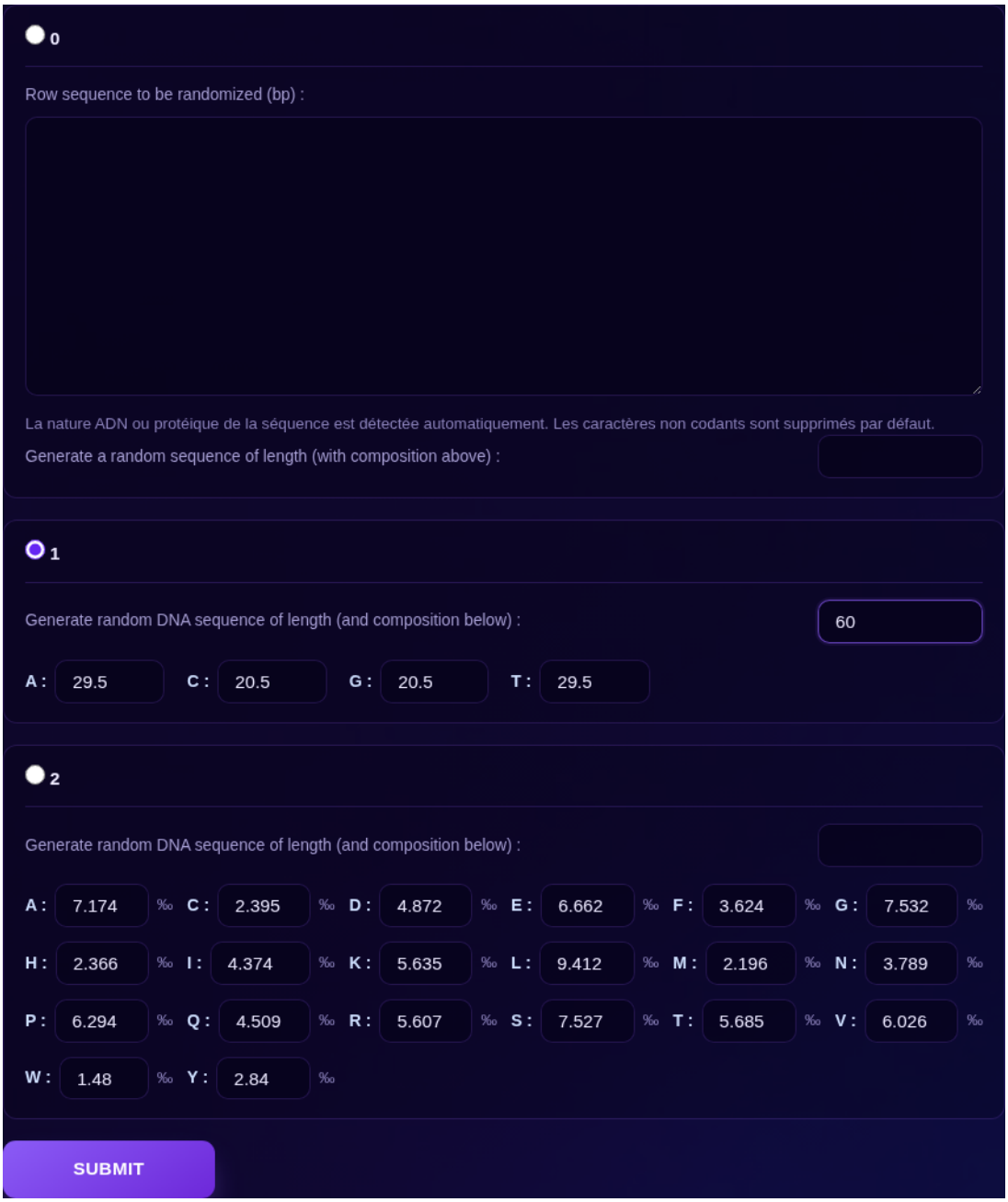

Figure 3.26: Random sequence generation form (28 fields, grouped into three methods).

Field	Label / translation	Type	Values, default, role
`procedure`	(choice of method) *Method*	radio	`fromseq` / `fromACGT` / `fromAA`.
`seq`	Row sequence to be randomized *Sequence to shuffle*	long text	Used with `fromseq`.
`length1`	Generate a random sequence of length (with composition above) *Length (same composition)*	text	Used with `fromseq`.

Field	Label / translation	Type	Values, default, role
`length2`	Generate random DNA sequence of length (and composition below) *Length of the DNA sequence*	text	Used with `fromACGT`.
`dnaA, dnaC, dnaG, dnaT`	A / C / G / T *A/C/G/T frequencies (‰)*	text	29.5 / 20.5 / 20.5 / 29.5 by default.
`length3`	(length for fromAA) *Length of the protein sequence*	text	Used with `fromAA`.
`a, c, d, e, f, g, h, i, k, l, m, n, p, q, r, s, t, v, w, y`	(20 amino acid frequencies) *Frequencies of the 20 amino acids (‰)*	text	Default values based on the human proteome (e.g. : L = 9.412, W = 1.48).

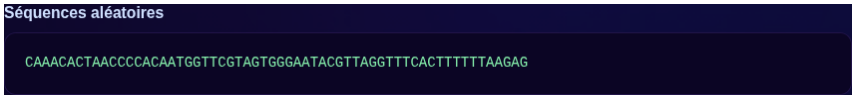

Figure 3.27: Random sequence generated.

Symfony implementation

```
public function randomSeqsAction(Request $request,
    RandomSequencesManager $manager)
{
    $result = "";
    $form = $this->createForm(RandomSequencesType::class);

    if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
        $formData = $form->getData();

        switch ($formData["procedure"]) {
            case "fromseq":
                $result = $manager->createFromSeq($formData["length1"],
                    $formData["seq"]);
                break;
            case "fromACGT":
                $freqs = ['A' => $formData["dnaA"], 'C' => $formData["
                    dnaC"],
                          'G' => $formData["dnaG"], 'T' => $formData["
                              dnaT"]];
                $result = $manager->createFromACGT($freqs, $formData["
                    length2"]);
                break;
            case "fromAA":
                $freqs = []; // the 20 frequencies, cf. formData['a'] ..
                    formData['y']
```

```
                $result = $manager->createFromAA($freqs, $formData["
                    length3"]);
                break;
            }
        }

        return $this->render('minitools/randomSeqs.html.twig',
            ['form' => $form->createView(), 'results' => $result]);
    }
```

Vanilla implementation

```
$manager = new RandomSequencesManager($nucleotidApi, $aminoApi);

$sequence = $manager->createFromACGT(
    ['A' => 29.5, 'C' => 20.5, 'G' => 20.5, 'T' => 29.5],
    60
);
```

3.4.14 Reduced protein alphabet

This tool groups the amino acids of a protein sequence into a more restricted alphabet, according to one of the published classifications (Murphy, Wang & Wang, Li *et al.*, IMGT...), or according to a custom alphabet supplied by the user.

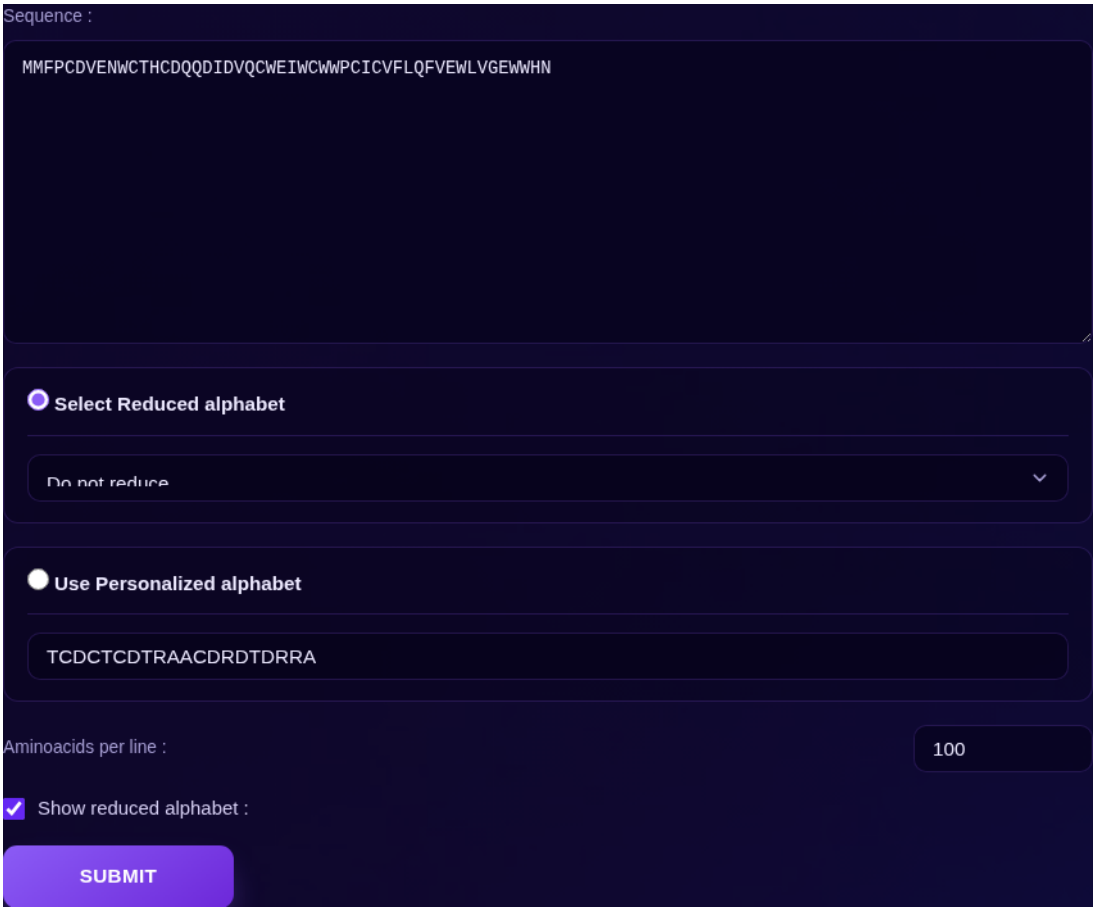

Figure 3.28: Protein alphabet reduction form.

Field	Label / translation	Type	Values, default, role
`seq`	Sequence *Protein sequence*	long text	`ARNDCEQGHILKMFPSTWYVX*` alphabet by default.
`mode`	Select Reduced alphabet / Use Personalized alphabet *Predefined or custom alphabet*	radio	`pre` or `custom`.
`type`	(list of reduced alphabets) *Reduction type*	drop-down list	Used with `mode=pre`: Murphy (2/4/8/10/15), Wang & Wang (2/3/5), Li *et al.* (3/4/5/10), IMGT (3/5/11), etc.
`custom_alphabet`	(20-letter string) *Custom alphabet*	text	Used with `mode=custom`; must be exactly 20 characters, one per amino acid of the `ARNDCEQGHILKMFPSTWYV` series.
`aaperline`	Aminoacids per line *Amino acids per line*	text	100 by default, for formatting the display.
`show_reduced`	Show reduced alphabet *Show reduced alphabet*	checkbox	Also displays the mapping table used.

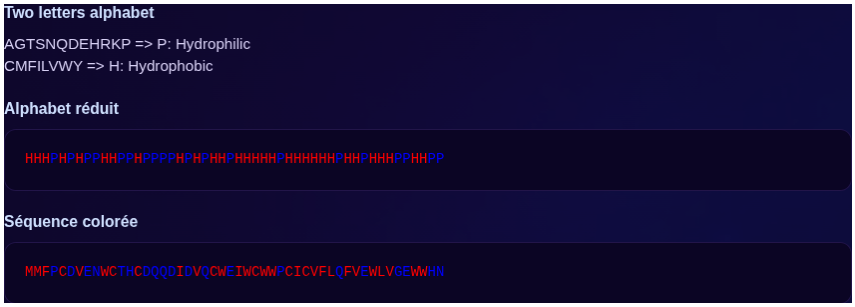

Figure 3.29: Reduced sequence, colored by alphabet letter.

Symfony implementation

```
public function reduceProteinAlphabetAction(Request $request,
    ReduceProteinAlphabetManager $manager)
{
    $sequence = $reducedSequence = "";
    $reduceCode = [];
    $form = $this->createForm(ReduceAlphabetType::class);

    if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
        $formData = $form->getData();
        $sequence = $formData["seq"];

        if ($formData["mode"] === "pre") {
            $reducedSequence = $manager->reduceAlphabet($sequence,
                $formData["type"]);
```

```
            if ($formData["show_reduced"]) {
                $reduceCode = $manager->createReduceCode($formData["type
                    "]);
            }
        } else {
            $reducedSequence = $manager->reduceAlphabetCustom($sequence,
                $formData["custom_alphabet"]);
        }
    }

    return $this->render('minitools/reduceProteinAlphabet.html.twig', [
        'form'      => $form->createView(),
        'sequence'  => $reducedSequence,
        'aaperline' => $formData['aaperline'] ?? 100,
        'reduce_code' => $reduceCode,
    ]);
}
```

Vanilla implementation

```
$manager = new ReduceProteinAlphabetManager($proteinColors,
    $proteinReductionApi);

// predefined alphabet (Murphy et al., 10 groups)
$reduced = $manager->reduceAlphabet('ARNDCEQGHILKMFPSTWYVX*', 'Murphy10'
    );

// custom alphabet (20 letters, one per amino acid of
    ARNDCEQGHILKMFPSTWYV)
$reducedCustom = $manager->reduceAlphabetCustom(
    'ARNDCEQGHILKMFPSTWYVX*',
    'TCDCTCDTRAACDRDTDRRA'
);
```

Note

Validation of the `custom_alphabet` field (exactly 20 characters) is only applied when `mode` is `custom`: choosing a predefined alphabet exempts you from supplying a correct 20-letter string.

3.4.15 Restriction enzyme digestion

For one or more DNA sequences, this tool searches for the cut sites of the full set (or a filtered subset) of known restriction endonucleases, and returns the number and position of the cuts for each.

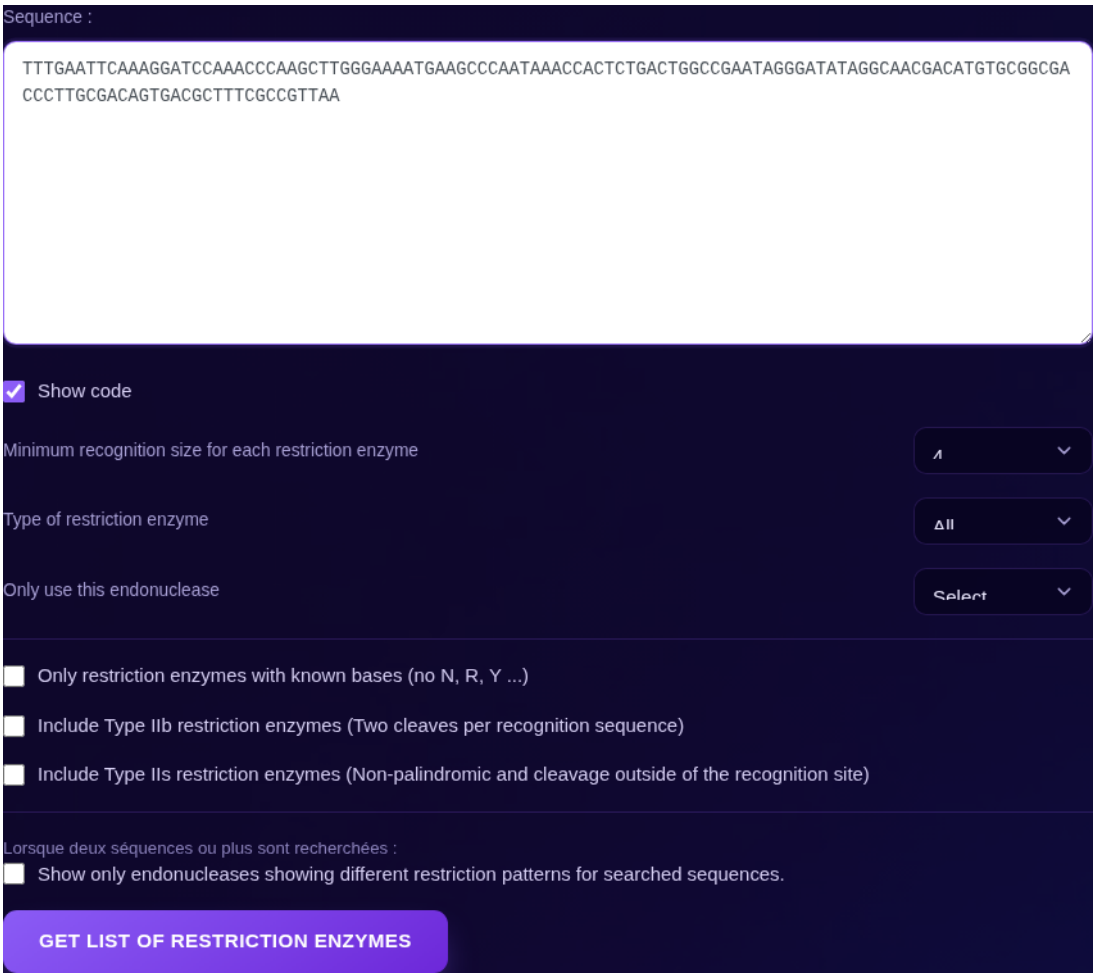

Figure 3.30: Restriction enzyme digestion form.

Field	Label / translation	Type	Values, default, role
`sequence`	Sequence *Sequence(s) to analyze*	long text	A single raw sequence, or several sequences in FASTA format (`>name`). 1,000,000 characters maximum.
`showcode`	Show code *Show code*	checkbox	Checked by default.
`minimum`	Minimum recognition size for each restriction enzyme *Minimum recognition site size*	drop-down list	1 to 8, 4 by default.
`retype`	Type of restriction enzyme *Cut type*	drop-down list	`All` (0) / `Blunt ends` (1) / `Overhang end` (2, sticky ends).
`wre`	Only use this endonuclease *Restrict to this enzyme*	drop-down list	Empty (= all enzymes) or the name of a specific enzyme.
`defined`	Only restriction enzymes with known bases (no N, R, Y ...) *Fully defined bases only*	checkbox	Excludes patterns containing degenerate bases.

Field	Label / translation	Type	Values, default, role
`IIb`	Include Type IIb restriction enzymes *Include Type IIb enzymes*	checkbox	Two cuts per recognition site.
`IIs`	Include Type IIs restriction enzymes *Include Type IIs enzymes*	checkbox	Non-palindromic cut, outside the recognition site.
`onlydiff`	Show only endonucleases showing different restriction patterns for searched sequences *Show only differing patterns*	checkbox	Only relevant with multiple input sequences.

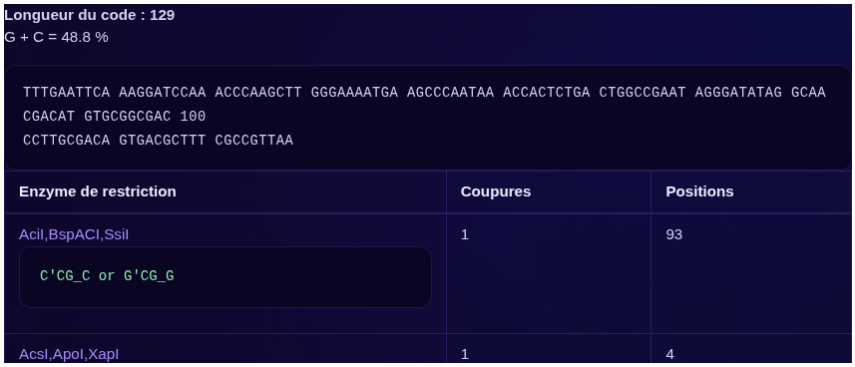

Figure 3.31: List of enzymes and their cut sites.

Symfony implementation

```
public function restrictionDigestAction(Request $request,
    RestrictionDigestManager $manager)
{
     $sequence = $digestion = $enzymes_array = $digestionMulti = [];
     $bShowCode = false;
     $form = $this->createForm(RestrictionEnzymeDigestType::class);

     if ($request->isMethod('POST') && $form->handleRequest($request)->
         isValid()) {
          $formData  = $form->getData();
          $bShowCode = $formData["showcode"];
          $sequence  = $manager->extractSequences($formData["sequence"]);

          $enzymes_array = $manager->getNucleolasesInfos(
              $formData["IIs"], $formData["IIb"], $formData["defined"]
          );
          $enzymes_array = $manager->reduceEnzymesArray(
              $enzymes_array, $formData["minimum"], $formData["retype"],
              $formData["defined"], $formData["wre"]
          );

          foreach ($sequence as $number => $val) {
              $digestion[$number] = $manager->restrictionDigest(
                  $enzymes_array, $sequence[$number]["seq"]);
          }
```

```
23
24          if (count($sequence) > 1) {
25              $digestionMulti = $manager->enzymesForMultiSeq(
26                  $sequence, $digestion, $enzymes_array, $formData["
                        onlydiff"], $formData["wre"]
27              );
28          }
29      }
30
31      return $this->render('minitools/restrictionDigest.html.twig', [
32          'form'            => $form->createView(),
33          'show_code'       => $bShowCode,
34          'sequence'        => $sequence,
35          'digestion'       => $digestion,
36          'digestion_multi' => array_unique($digestionMulti),
37          'enzymes_array'   => $enzymes_array,
38      ]);
39  }
40
41  // secondary route: /minitools/show-vendors/{enzyme}, called via Ajax
        from the
42  // template to display the commercial vendors of a given enzyme
43  public function showVendorsAction($enzyme, RestrictionDigestManager
        $manager)
44  {
45      $message = "";
46      $enzyme_array = $manager->getVendors($message, $enzyme);
47      return new JsonResponse(["message" => $message, "vendors" =>
            $enzyme_array]);
48  }
```

Vanilla implementation

```
1  $manager = new RestrictionDigestManager(
2      $vendorLinksApi, $typeIIEndonucleaseApi, $typeIIbEndonucleaseApi,
3      $typeIIsEndonucleaseApi, $vendorApiAdapter
4  );
5
6  $sequence = $manager->extractSequences(">seq1\
       nACGTACGTACGTTAGCTAGCTAGCTAGC");
7  $enzymes  = $manager->getNucleolasesInfos(false, false, false);
8  $enzymes  = $manager->reduceEnzymesArray($enzymes, 4, 0, false, "");
9
10 $digestion = $manager->restrictionDigest($enzymes, $sequence[0]["seq"]);
```

Note

The `/minitools/show-vendors/{enzyme}` route is a special case: it is the only BioTools route that does not render a form, but responds in JSON. In the demo, it is called in the background (Ajax) when the user clicks on an enzyme's name in the results, to display its commercial vendors without reloading the page.

3.4.16 Sequence alignment

This tool aligns two sequences (DNA/RNA or protein, detected automatically) using the Smith-Waterman algorithm, and displays the result with the gaps needed for the alignment.

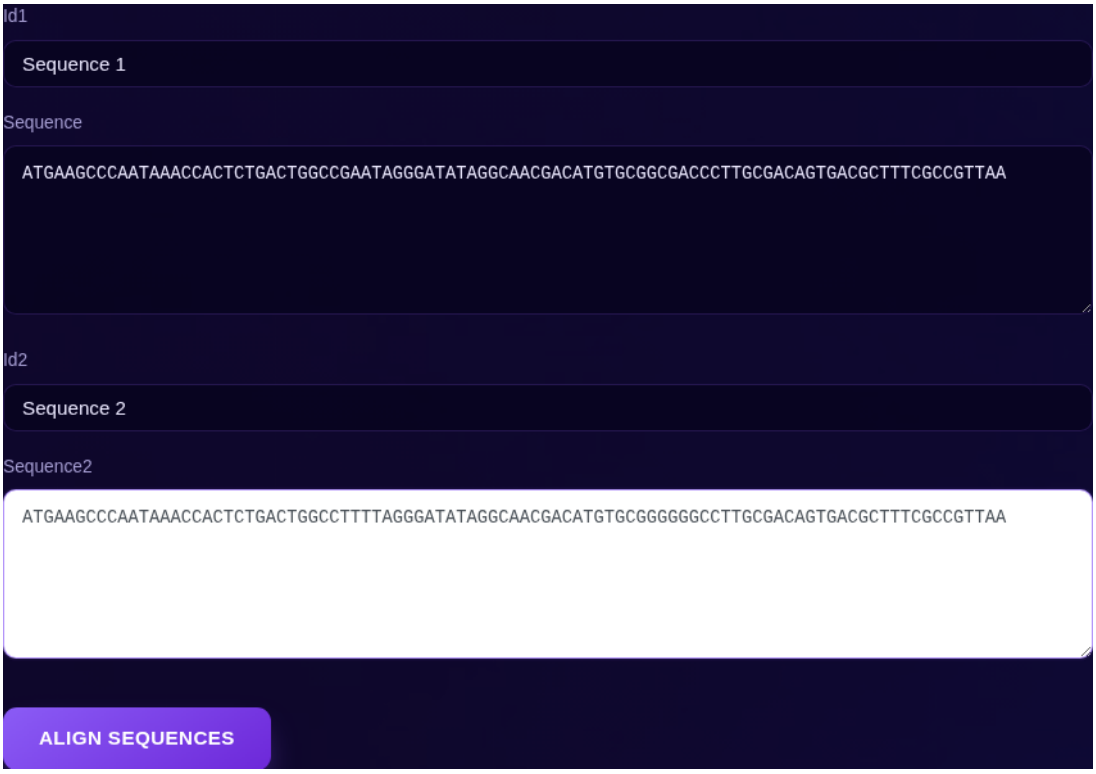

Figure 3.32: Two-sequence alignment form.

Field	Label / translation	Type	Values, default, role
`id1`	(name of the first sequence) *Sequence 1 name*	text	"Sequence 1" by default.
`sequence`	(first sequence) *Sequence 1*	long text	Cleaned automatically (uppercase, U→T, X→N).
`id2`	(name of the second sequence) *Sequence 2 name*	text	"Sequence 2" by default.
`sequence2`	(second sequence) *Sequence 2*	long text	Same cleanup as `sequence`.

Warning

The combined length of the two sequences is limited (300 nucleotides in the demo): the algorithm builds a matrix whose size grows with the square of the sequences' length, which could otherwise exceed the memory or execution time allotted to PHP.

Figure 3.33: Aligned sequences, with the comparison line (|) between identical positions.

Symfony implementation

```php
public function seqAlignmentAction(Request $request,
    SequenceAlignmentManager $manager)
{
    $form = $this->createForm(SequenceAlignmentType::class);
    $sCompare = $sAlignSeqA = $sAlignSeqB = "";
    $id1 = $id2 = "";

    if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
        $formData = $form->getData();
        $id1 = $formData["id1"];
        $id2 = $formData["id2"];

        // DNA/protein detection: DNA if A+C+G+T >= half the sequence
        $countACGT = substr_count($formData["sequence"], "A")
            + substr_count($formData["sequence"], "C")
            + substr_count($formData["sequence"], "G")
            + substr_count($formData["sequence"], "T");

        if ($countACGT > (strlen($formData["sequence"]) / 2)) {
            $aAlignment = $manager->alignDNA($formData["sequence"],
                $formData["sequence2"]);
        } else {
            $aAlignment = $manager->alignProteins($formData["sequence"],
                $formData["sequence2"]);
        }

        $sAlignSeqA = $aAlignment["seqa"];
        $sAlignSeqB = $aAlignment["seqb"];
        $sCompare   = $manager->compareAlignment($sAlignSeqA,
            $sAlignSeqB);
    }

    return $this->render('minitools/seqAlignment.html.twig', [
        'form'       => $form->createView(),
        'compare'    => $sCompare,
        'align_seqa' => $sAlignSeqA,
        'align_seqb' => $sAlignSeqB,
```

```
        'sequence1'  => $id1,
        'sequence2'  => $id2,
    ]);
}
```

Vanilla implementation

```
$manager = new SequenceAlignmentManager($pam250MatrixApi);

$result   = $manager->alignDNA($sequenceA, $sequenceB);
$compare  = $manager->compareAlignment($result["seqa"], $result["seqb"])
    ;
```

Note

The Twig filter `compare_alignment` introduced in §3.3 is only a thin wrapper: it directly calls `SequenceAlignmentManager::compareAlignment()`, the same method used here on the controller side to build the comparison line before rendering.

3.4.17 Sequence manipulation

This tool applies a simple operation to a DNA/RNA sequence (removing non-coding characters, reversing the sequence, complement, reverse complement, displaying both strands, conversion to RNA), and can additionally compute its nucleotide composition and G+C content.

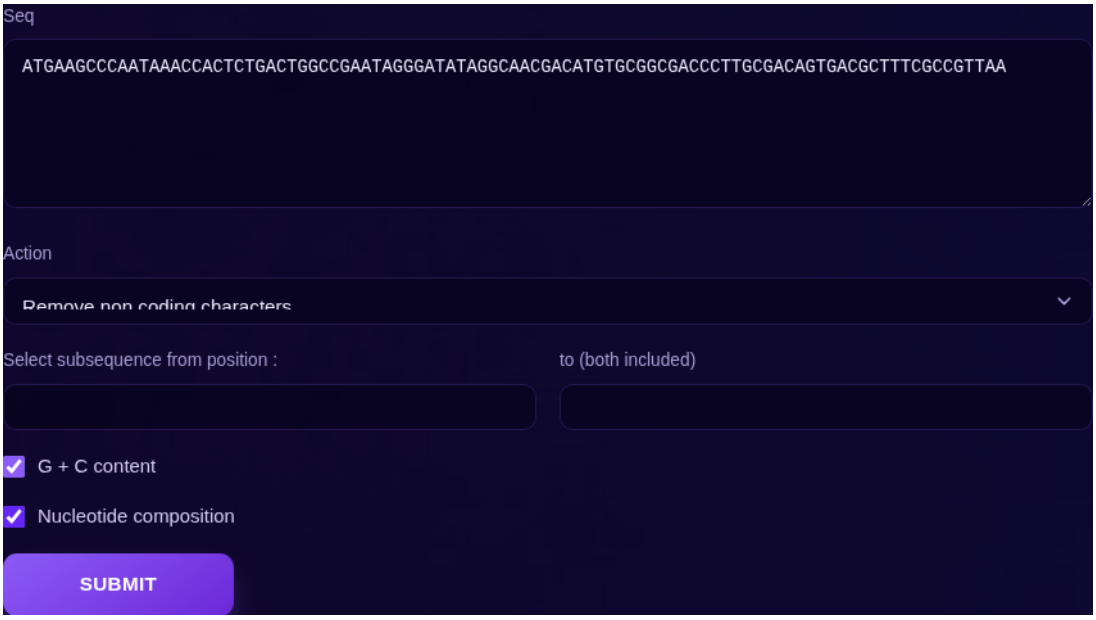

Figure 3.34: Sequence manipulation form.

Field	Label / translation	Type	Values, default, role
`seq`	(sequence) *Sequence to process*	long text	Cleaned automatically on submission (uppercase, X→N, non-coding characters stripped).
`action`	(list of operations) *Operation to apply*	single-choice list	`remove_non_coding` / `reverse` / `complement` / `reverse_and_complement` / `display_both_strands` / `toRNA`.
`start`	Select subsequence from position *Start of the subsequence*	text	1-indexed position; empty = start of the sequence.
`end`	to (both included) *End of the subsequence (inclusive)*	text	Empty = end of the sequence.
`GC`	G + C content *G+C content*	checkbox	Adds the G+C percentage to the result.
`ACGT`	Nucleotide composition *Nucleotide composition*	checkbox	Adds the count of each base (including any IUPAC degenerate bases present) to the result.

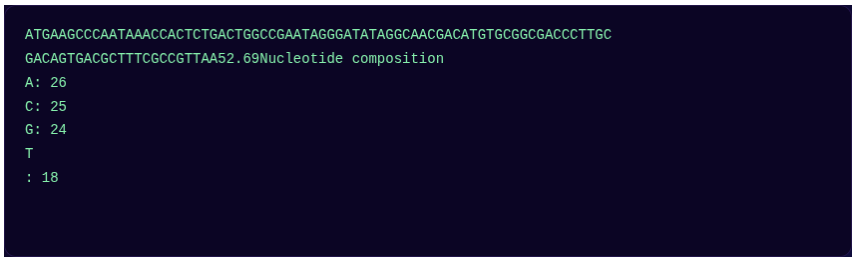

Figure 3.35: Result of the chosen operation.

Symfony implementation

```
public function sequencesManipulationAndDataAction(Request $request,
    SequenceManipulationAndDataManager $manager)
{
    $result = "";
    $form = $this->createForm(SequenceManipulationType::class);

    if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
        $formData = $form->getData();
        $seq = $formData["seq"];

        // optional subsequence
        if (isset($formData["start"]) || isset($formData["end"])) {
            $start = $formData["start"] != "" ? ($formData["start"] - 1)
                : 0;
            $end   = $formData["end"] != "" ? $formData["end"] : strlen(
                $formData["seq"]);
```

```
            $seq    = substr($formData["seq"], $start, $end - $start);
        }

        switch ($formData["action"]) {
            case "reverse":
                $result = strrev($seq);
                break;
            case "complement":
                $result = $manager->compDNA($seq);
                break;
            case "reverse_and_complement":
                $result = $manager->compDNA(strrev($formData["seq"]));
                break;
            case "display_both_strands":
                $result = $manager->displayBothStrands($seq);
                break;
            case "toRNA":
                $result = $manager->toRNA($seq);
                break;
            default:
                $result = $seq; // remove_non_coding: cleanup already
                    happened in PRE_SUBMIT
                break;
        }

        if ($formData["GC"]) {
            $result .= $manager->gcContent($seq);
        }
        if ($formData["ACGT"]) {
            $result .= $manager->acgtContent($seq);
        }
    }

    return $this->render('minitools/sequencesManipulationAndData.html.
        twig',
        ['form' => $form->createView(), 'result' => $result]);
}
```

Vanilla implementation

```
$manager = new SequenceManipulationAndDataManager();

$reverseComplement = $manager->compDNA(strrev('ATGCGGATCC'));
$gc                = $manager->gcContent('ATGCGGATCC');
```

Note

The `reverse`, `complement` and `reverse_and_complement` operations are not methods of `SequenceManipulationAndDataManager`: the first is a plain PHP `strrev()`, the other two go through `compDNA()`, a method inherited from BioPHP's `SequenceTrait` (§2.7) and not specific to BioTools.

3.4.18 Skews generator

This tool scans a sequence using a sliding window and plots, for each position, one or more composition indicators (GC-skew, AT-skew, KETO-skew, G+C content, oligo-skew) as an SVG chart.

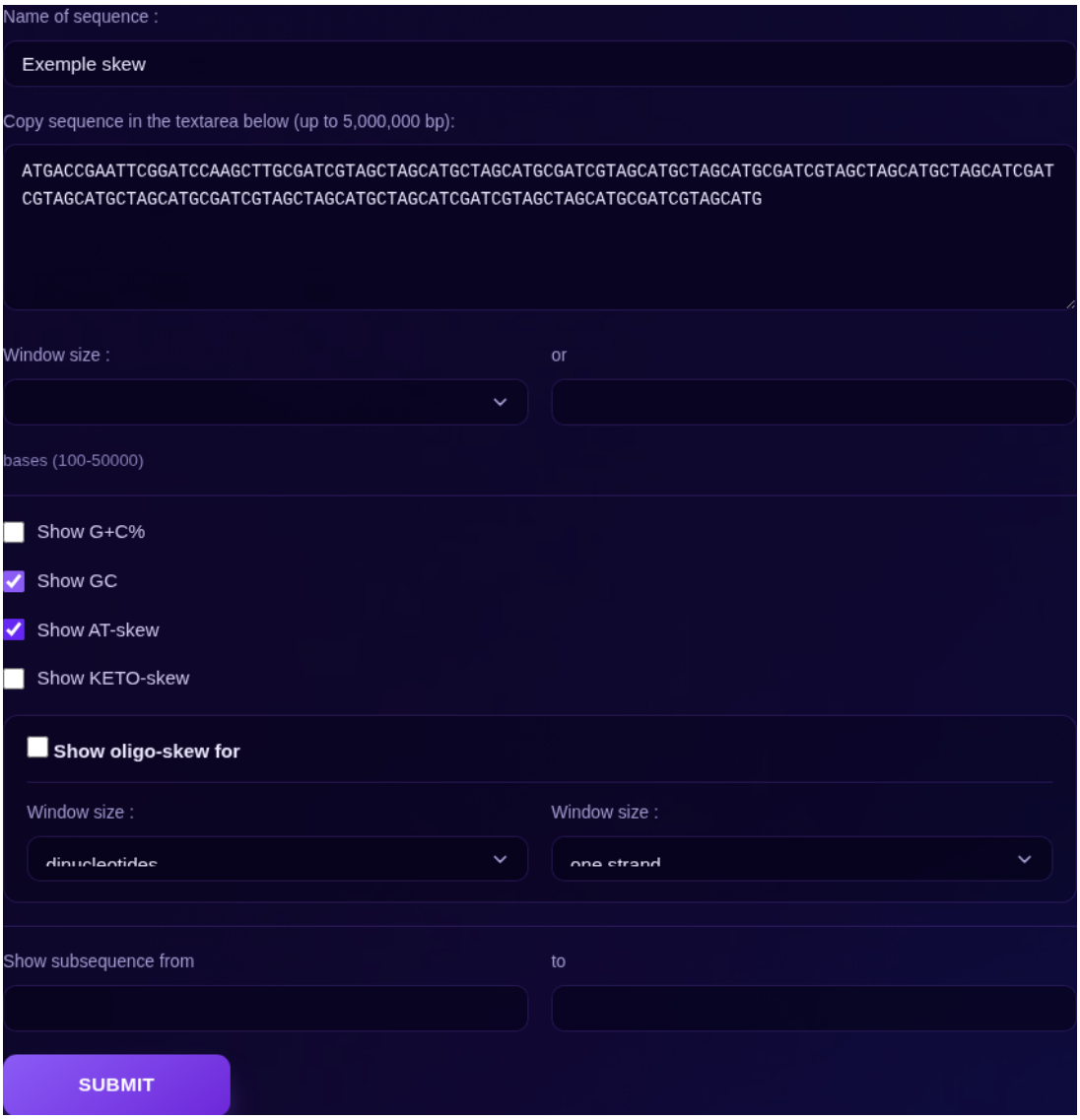

Figure 3.36: Skews generator form.

Field	Label / translation	Type	Values, default, role
`name`	Name of sequence *Sequence name*	text	Empty → "sequence" by default; used in the chart title.
`seq`	Copy sequence in the textarea below *Sequence to analyze*	long text	Up to 10,000,000 characters; cleaned automatically.
`window`	Window size *Window size (list)*	drop-down list	5,000 / 10,000 / 50,000 / 100,000.

Field	Label / translation	Type	Values, default, role
`window2`	or *Window size (custom)*	text	100 to 100,000; takes priority over `window` if filled in.
`GmC`	Show G+C% *Show G+C content*	checkbox	
`GC`	Show GC *Show GC-skew*	checkbox	
`AT`	Show AT-skew *Show AT-skew*	checkbox	
`KETO`	Show KETO-skew *Show KETO-skew*	checkbox	
`oskew`	Show oligo-skew for *Show oligo-skew*	checkbox	Enables the potentially long computation of `oligo_len`/`strands`.
`oligo_len`	dinucleotides. . . hexanucleotides *Oligonucleotide length*	drop-down list	2 to 6.
`strands`	one strand / both strands *One or both strands*	drop-down list	Changes the distance formula used (Almeida *et al.* for both strands, standard Pearson otherwise).
`from` / `to`	Show subsequence from / to *Subsequence to analyze*	text	Optional; restrict the analysis to a range.

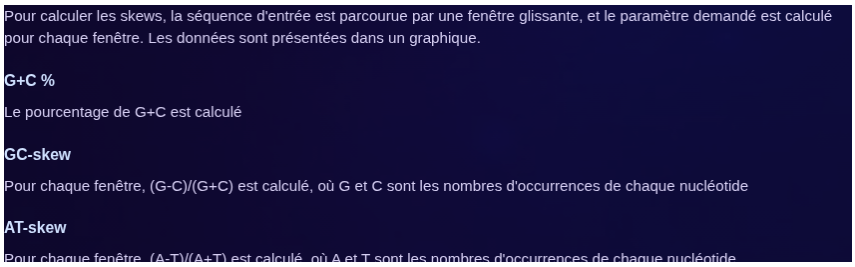

Figure 3.37: SVG chart of the requested skews.

Symfony implementation

```
public function skewsAction(Request $request, SkewsManager $manager)
{
    $imageResult = "";
    $form = $this->createForm(SkewsType::class);

    if ($request->isMethod('POST') && $form->handleRequest($request)->
        isValid()) {
        $formData = $form->getData();
        $sequence = $formData["seq"];

        if ($formData["from"] || $formData["to"]) {
            if ($manager->strIsInt($formData["to"])) {
                $sequence = substr($formData["seq"], 0, $formData["to"])
                    ;
```

```
            }
            if ($manager->strIsInt($formData["from"])) {
                $sequence = substr($formData["seq"], $formData["from"]);
            }
        }

        $aOligoSkew = [];
        if ($formData["oskew"] && $manager->strIsInt($formData["
            oligo_len"])) {
            $aOligoSkew = $manager->oligoSkewArrayCalculation(
                $sequence, $formData["window"], $formData["oligo_len"],
                    $formData["strands"]
            );
        }

        $imageResult = basename($manager->createImage(
            $sequence, $formData["window"], $formData["GC"], $formData["
                AT"],
            $formData["KETO"], $formData["GmC"], $aOligoSkew,
            $formData["oligo_len"], $formData["from"], $formData["to"],
                $formData["name"]
        ));
    }

    return $this->render('minitools/skews.html.twig',
        ['form' => $form->createView(), 'image' => $imageResult]);
}
```

Vanilla implementation

```
$manager = new SkewsManager($nucleotidsGraphsConfig, $oligosManager);

$image = $manager->createImage(
    $sequence, 5000, true, true, false, false,
    [], 2, '', '', 'ma_sequence'
);
// $image holds the full path of the SVG file written to disk
```

Warning

As with the chaos game representation and the distance between sequences, `createImage()` writes an SVG file to the directory configured by `path_graphs` (§3.2). A missing or non-writable folder results in a server error; see the detailed write mechanism in §3.5.

Note

The `oskew` option (oligo-skew) is the most computationally expensive: window by window, it compares the oligonucleotide frequencies of the entire sequence to those of each window, using the method of Almeida *et al.* (2001) on both strands, or a classic Pearson distance on a single strand.

3.4.19 Useful formulas

This is not, strictly speaking, a sequence analysis tool, but a toolbox of common laboratory formulas: molecular weight of nucleic acids, pmol/µg conversions, temperature, pressure and centrifugation.

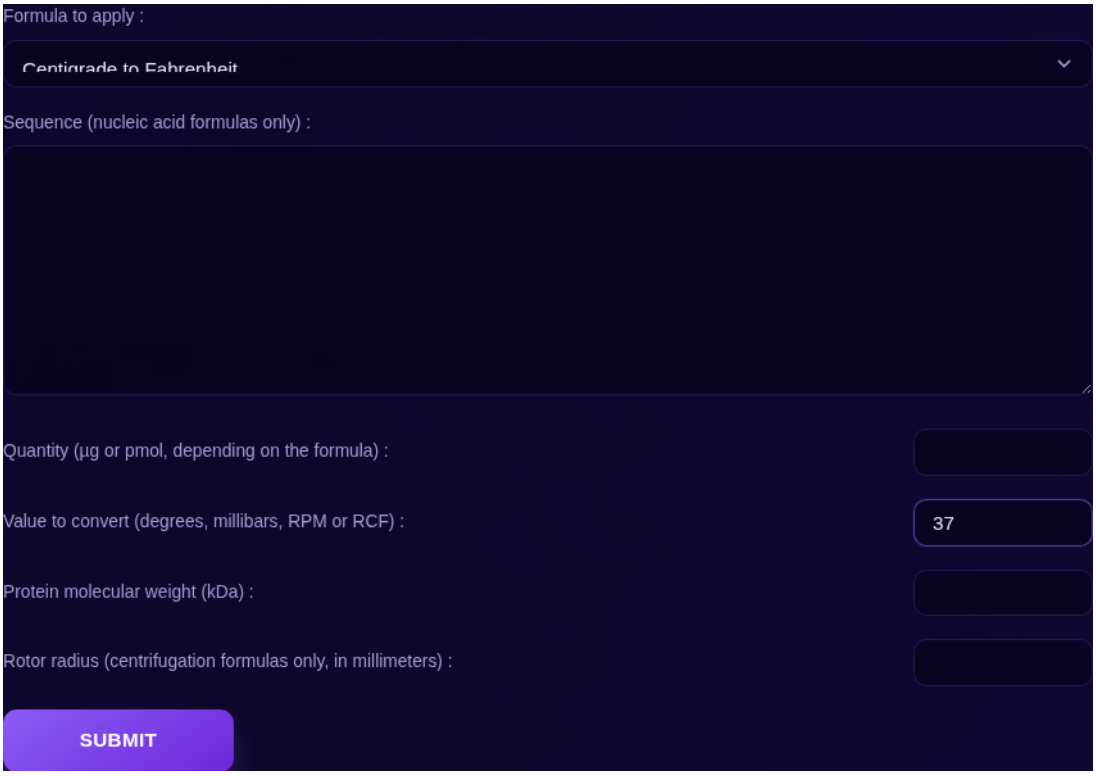

Figure 3.38: Useful formulas form.

Field	Label / translation	Type	Values, default, role
`formula`	Formula to apply *Formula to apply*	grouped drop-down list	23 formulas grouped into 5 categories: DNA, RNA, temperature/pressure, proteins, centrifugation.
`sequence`	Sequence (nucleic acid formulas only) *Sequence (nucleic acid formulas only)*	long text	Cleaned automatically; its length is used as the number of bases in DNA/RNA formulas.
`quantity`	Quantity (µg or pmol, depending on the formula) *Quantity (µg or pmol)*	text	Used by the pmol↔µg conversion formulas.
`value`	Value to convert (degrees, millibars, RPM or RCF) *Value to convert*	text	Used by the temperature, pressure and centrifugation formulas.
`molecularWeight`	Protein molecular weight (kDa) *Protein molecular weight (kDa)*	text	Used by the protein formulas.
`radius`	Rotor radius (centrifugation formulas only, in millimeters) *Rotor radius (mm)*	text	Used by the centrifugation formulas.

```
98.6000
```

Figure 3.39: Result of the chosen formula.

Symfony implementation

```php
public function usefulFormulasAction(Request $request, FormulasManager $manager)
{
    $result = $formula = null;
    $form = $this->createForm(FormulasType::class);

    if ($request->isMethod('POST') && $form->handleRequest($request)->isValid()) {
        $formData = $form->getData();
        $formula  = $formData["formula"];

        $result = match ($formula) {
            "mw_dsdna"       => $manager->mwOfDsDNA($formData["sequence"]),
            "centi_to_fahren" => $manager->centiToFahren($formData["value"]),
            "mbar_to_inchhg" => $manager->mbarToInchHg($formData["value"]),
            "rpm_to_rcf"     => $manager->rpmToRcf($formData["value"], $formData["radius"]),
            // ... 19 other cases, one per entry in the drop-down list
            default => null,
        };
    }

    return $this->render('minitools/usefulFormulas.html.twig',
        ['form' => $form->createView(), 'formula' => $formula, 'result' => $result]);
}
```

Vanilla implementation

```php
$manager = new FormulasManager();

$fahrenheit = $manager->centiToFahren(37);        // 98.6
$inchHg     = $manager->mbarToInchHg(1013.25);    // standard atmospheric pressure
```

Warning

Two of the 23 formulas were corrected relative to the biophp.org reference code, which contained a mathematical error:

- **Centigrade to Fahrenheit**: the reference code applied `32 + (C × 0.555)`, whereas 0.555 (≈5/9) is the Fahrenheit→Centigrade factor used in the wrong direction (100°C gave 87.5°F instead of 212°F). BioTools applies `32 + (C × 1.8)`.

- **Millibars to inches of mercury**: the reference code applied `mbar × 0.0394` (the millimeter→inch factor), overestimating the result by about 33 %. BioTools applies `mbar × 0.02953`.

If a calculation based on these two formulas does not match a result found elsewhere on the web, this correction is very likely the reason.

3.5 SVG graphics

Three tools in this chapter produce a chart: the chaos game representation (§3.4.1, in both CGR and FCGR mode), the distance between sequences (§3.4.2, dendrogram) and the skews generator (§3.4.18). All three rely on the same building block, `Amelaye\BioTools\Service\Graphics\SvgCanvas`.

3.5.1 SvgCanvas

BioTools does not depend on PHP's GD extension: `SvgCanvas` rewrites in SVG the handful of drawing primitives that the graphics managers need, with a direct correspondence to the GD functions it replaces.

Method	Replaces (GD)	Role
`__construct($width, $height)`	`imagecreate()`	Creates an SVG canvas of the given size.
`rgb($r, $g, $b)` *static*	`imagecolorallocate()`	Builds a `#rrggbb` color from three 0–255 components.
`background($couleur)`	(background color allocation)	Fills the whole canvas.
`rect($x1, $y1, $x2, $y2, $couleur)`	`imagefilledrectangle()`	Filled rectangle; the coordinates can be given in any order.
`line($x1, $y1, $x2, $y2, $couleur)`	`imageline()`	Draws a line segment.
`pixel($x, $y, $couleur)`	`imagesetpixel()`	A single point.
`pixelCloud($points, $couleur)`	(loop over `imagesetpixel()`)	Groups a series of points into a single SVG path instead of one element per point — this is what avoids excessively large files for the skews and the dendrogram.
`text($police, $x, $y, $texte, $couleur)`	`imagestring()`	Writes text; the five GD font sizes (1 to 5) are approximated by equivalent SVG sizes and baselines.
`render()`	—	Returns the complete SVG document as a string.
`save($chemin)`	`imagepng($im, $chemin)`	Writes the SVG to the given path and returns that path.

Note

The `$police` parameter of `text()` is not a real font: it is the integer 1 to 5 of the five bitmap fonts built into GD (`imagestring($im, $police, ...)`), kept so that code ported from GD does not have to change its calls; `SvgCanvas` translates this integer into an SVG font size and baseline offset close to the original bitmap rendering.

3.5.2 Where the files go

The three managers involved (`ChaosGameRepresentationManager`, in both CGR and FCGR mode, `DistanceAmongSequencesManager` and `SkewsManager`) all call `SvgCanvas::save()` with a path built from the same configuration parameter, `amelaye_biotools.nucleotids_graphs.path_graphs` (§3.2). The file name includes a random suffix (`bin2hex(random_bytes(4))`) so that two simultaneous requests do not overwrite the same file.

Warning

This directory must satisfy two conditions at once, which are easy to miss separately:

- **writable** by the PHP process (the typical server-side error looks like `file_put_contents(): failed to open stream`);
- **served publicly** by the web server, since the controller only returns the file's *name* (`basename(...)`) — it is up to the template to build the public URL to that same directory.

This is exactly the issue encountered and fixed on the demo site while producing the screenshots for this chapter (§3.4.2, §3.4.1): a non-writable directory shows up client-side as a generic 500 error, with no further indication.

3.6 Summary

Tool	Form	Manager	Main method
Chaos Game Representation (CGR/FCGR) §3.4.1	`ChaosGameRepresentationType`	`ChaosGameRepresentationManager`	`cgrCompute()` / `fcgrCompute()`
Distance between sequences §3.4.2	`DistanceAmongSequencesType`	`DistanceAmongSequencesManager`	`upgmaClustering()`
DNA→protein translation §3.4.3	`DnaToProteinType`	`DnaToProteinManager`	`translateDNAToProtein()`
Palindromic sequences §3.4.4	`FindPalindromesType`	`FindPalindromeManager`	`findPalindromicSeqs()`
GC content (FASTA upload) §3.4.5	`FastaUploaderType`	`FastaUploaderManager`	`createFiles()`
Melting temperature §3.4.6	`MeltingTemperatureType`	`MeltingTemperatureManager`	`basicCalculations()` / `neighborCalculations()`
Microarray data analysis §3.4.7	`MicroArrayDataAnalysisType`	`MicroarrayAnalysisAdaptiveManager`	`processMicroarrayDataAdaptiveQuantificationMethod()`
Microsatellite repeats §3.4.8	`MicrosatelliteRepeatsFinderType`	`MicrosatelliteRepeatsFinderManager`	`findMicrosatelliteRepeats()`
Oligonucleotide frequency §3.4.9	`OligoNucleotideFrequencyType`	`OligosManager` *(BioPHP)*	`findOligos()`
PCR amplification §3.4.10	`PcrAmplificationType`	`PcrAmplificationManager`	`amplify()`
Protein properties §3.4.11	`ProteinPropertiesType`	`ProteinPropertiesManager`	`aminoacidContent()`, `proteinMolecularWeight()`…
Protein→DNA back-translation §3.4.12	`ProteinToDnaType`	`ProteinToDnaManager`	`translateProteinToDNA()`
Random sequences §3.4.13	`RandomSequencesType`	`RandomSequencesManager`	`createFromSeq()` / `createFromACGT()` / `createFromAA()`
Reduced protein alphabet §3.4.14	`ReduceAlphabetType`	`ReduceProteinAlphabetManager`	`reduceAlphabet()` / `reduceAlphabetCustom()`
Restriction enzyme digestion §3.4.15	`RestrictionEnzymeDigestType`	`RestrictionDigestManager`	`restrictionDigest()`

Tool	Form	Manager	Main method
Sequence alignment §3.4.16	`SequenceAlignmentType`	`SequenceAlignmentManager`	`alignDNA()` / `alignProteins()`
Sequence manipulation §3.4.17	`SequenceManipulationType`	`SequenceManipulationAndDataManager`	`displayBothStrands()`, `gcContent()`...
Skews generator §3.4.18	`SkewsType`	`SkewsManager`	`oligoSkewArrayCalculation()`, `createImage()`
Useful formulas §3.4.19	`FormulasType`	`FormulasManager`	23 conversion methods