Server-side web frameworks

The previous article showed you what the communication between web clients and servers looks like, the nature of HTTP requests and responses, and what a server-side web application needs to do in order to respond to requests from a web browser. With this knowledge under our belt, it's time to explore how web frameworks can simplify these tasks, and give you an idea of how you'd choose a framework for your first server-side web application.

Prerequisites:	Basic computer literacy. Basic understanding of how server-side code handles and responds to HTTP requests (see Client-Server overview).
Objective:	To understand how web frameworks can simplify development/maintenance of server-side code and to get readers thinking about selecting a framework for their own development.

The following sections illustrate some points using code fragments taken from real web frameworks. Don't be concerned if it doesn't **all** make sense now; we'll be working you through the code in our framework-specific modules.

Overview

Server-side web frameworks (a.k.a. "web application frameworks") are software frameworks that make it easier to write, maintain and scale web applications. They provide tools and libraries that simplify common web development tasks, including routing URLs to appropriate handlers, interacting with databases, supporting sessions and user authorization, formatting output (e.g. HTML, JSON, XML), and improving security against web attacks.

The next section provides a bit more detail about how web frameworks can ease web application development. We then explain some of the criteria you can use for choosing a web framework, and then list some of your options.

What can a web framework do for you?

Web frameworks provide tools and libraries to simplify common web development operations. You don't *have* to use a server-side web framework, but it is strongly advised — it will make your life a lot easier.

This section discusses some of the functionality that is often provided by web frameworks (not every framework will necessarily provide all of these features!).

Work directly with HTTP requests and responses

As we saw in the last article, web servers and browsers communicate via the HTTP protocol — servers wait for HTTP requests from the browser and then return information in HTTP responses. Web frameworks allow you to write simplified syntax that will generate server-side code to work with these requests and responses. This means that you will have an easier job, interacting with easier, higher-level code rather than lower level networking primitives.

The example below shows how this works in the Django (Python) web framework. Every "view" function (a request handler) receives an `HttpRequest` object containing request information, and is required to return an `HttpResponse` object with the formatted output (in this case a string).

```
# Django view function
from django.http import HttpResponse

def index(request):
    # Get an HttpRequest (request)
    # perform operations using information from the request.
    # Return HttpResponse
    return HttpResponse('Output string to return')
```

Route requests to the appropriate handler

Most sites will provide a number of different resources, accessible through distinct URLs. Handling these all in one function would be hard to maintain, so web frameworks provide simple mechanisms to map URL patterns to specific handler functions. This approach also has benefits in terms of maintenance, because you can change the URL used to deliver a particular feature without having to change the underlying code.

Different frameworks use different mechanisms for the mapping. For example, the Flask (Python) web framework adds routes to view functions using a decorator.

```
@app.route("/")
def hello():
    return "Hello World!"
```

While Django expects developers to define a list of URL mappings between a URL pattern and a view function.

```
urlpatterns = [
```

```
    url(r'^$', views.index),
    # example: /best/myteamname/5/
    url(r'^best/(?P<team_name>\w.+?)/(?P<team_number>[0-9]+)/$', views.best)
]
```

Make it easy to access data in the request

Data can be encoded in an HTTP request in a number of ways. An HTTP `GET` request to get files or data from the server may encode what data is required in URL parameters or within the URL structure. An HTTP `POST` request to update a resource on the server will instead include the update information as "POST data" within the body of the request. The HTTP request may also include information about the current session or user in a client-side cookie.

Web frameworks provide programming-language-appropriate mechanisms to access this information. For example, the `HttpRequest` object that Django passes to every view function contains methods and properties for accessing the target URL, the type of request (e.g. an HTTP `GET`), `GET` or `POST` parameters, cookie and session data, etc. Django can also pass information encoded in the structure of the URL by defining "capture patterns" in the URL mapper (see the last code fragment in the section above).

Abstract and simplify database access

Websites use databases to store information both to be shared with users, and about users. Web frameworks often provide a database layer that abstracts database read, write, query, and delete operations. This abstraction layer is referred to as an Object-Relational Mapper (ORM).

Using an ORM has two benefits:

- You can replace the underlying database without necessarily needing to change the code that uses it. This allows developers to optimize for the characteristics of different databases based on their usage.
- Basic validation of data can be implemented within the framework. This makes it easier and safer to check that data is stored in the correct type of database field, has the correct format (e.g. an email address), and isn't malicious in any way (hackers can use certain patterns of code to do bad things such as deleting database records).

For example, the Django web framework provides an ORM, and refers to the object used to define the structure of a record as the *model*. The model specifies the field *types* to be stored, which may provide field-level validation on what information can be stored (e.g. an email field would only allow valid email addresses). The field definitions may also specify their maximum size, default values, selection list options, help text for documentation, label text for forms etc. The model doesn't state any information about the underlying database as that is a configuration setting that may be changed separately of our code.

The first code snippet below shows a very simple Django model for a `Team` object. This stores the team name and team level as character fields and specifies a maximum number of characters to be stored for each record. The `team_level` is a choice field, so we also provide a mapping between choices to be displayed and data to be stored, along with a default value.

```
#best/models.py

from django.db import models

class Team(models.Model):
    team_name = models.CharField(max_length=40)

    TEAM_LEVELS = (
        ('U09', 'Under 09s'),
        ('U10', 'Under 10s'),
        ('U11', 'Under 11s'),
        ...  #list our other teams
    )
    team_level = models.CharField(max_length=3,choices=TEAM_LEVELS,default='
```

The Django model provides a simple query API for searching the database. This can match against a number of fields at a time using different criteria (e.g. exact, case-insensitive, greater than, etc.), and can support complex statements (for example, you can specify a search on U11 teams that have a team name that starts with "Fr" or ends with "al").

The second code snippet shows a view function (resource handler) for displaying all of our U09 teams. In this case we specify that we want to filter for all records where the `team_level` field has exactly the text 'U09' (note below how this criteria is passed to the `filter()` function as an argument with field name and match type separated by double underscores: **team_level__exact**).

```
#best/views.py

from django.shortcuts import render
from .models import Team

def youngest(request):
    list_teams = Team.objects.filter(team_level__exact="U09")
    context = {'youngest_teams': list_teams}
    return render(request, 'best/index.html', context)
```

Rendering data

Web frameworks often provide templating systems. These allow you to specify the structure of

an output document, using placeholders for data that will be added when a page is generated. Templates are often used to create HTML, but can also create other types of document.

Web frameworks often provide a mechanism to make it easy to generate other formats from stored data, including JSON and XML.

For example, the Django template system allows you to specify variables using a "double-handlebars" syntax (e.g. `{{ variable_name }}`), which will be replaced by values passed in from the view function when a page is rendered. The template system also provides support for expressions (with syntax: `{% expression %}`), which allow templates to perform simple operations like iterating list values passed into the template.

> **Note:** Many other templating systems use a similar syntax, e.g.: Jinja2 (Python), handlebars (JavaScript), moustache (JavaScript), etc.

The code snippet below shows how this works. Continuing the "youngest team" example from the previous section, the HTML template is passed a list variable called `youngest_teams` by the view. Inside the HTML skeleton we have an expression that first checks if the `youngest_teams` variable exists, and then iterates it in a `for` loop. On each iteration the template displays the team's `team_name` value in a list item.

```
#best/templates/best/index.html

<!DOCTYPE html>
<html lang="en">
<body>

 {% if youngest_teams %}
    <ul>
    {% for team in youngest_teams %}
        <li>{{ team.team_name }}</li>
    {% endfor %}
    </ul>
{% else %}
    <p>No teams are available.</p>
{% endif %}

</body>
</html>
```

How to select a web framework

Numerous web frameworks exist for almost every programming language you might want to use (we list a few of the more popular frameworks in the following section). With so many choices, it can become difficult to work out what framework provides the best starting point for

your new web application.

Some of the factors that may affect your decision are:

- **Effort to learn:** The effort to learn a web framework depends on how familiar you are with the underlying programming language, the consistency of its API, the quality of its documentation, and the size and activity of its community. If you're starting from absolutely no programming experience then consider Django (it is one of the easiest to learn based on the above criteria). If you are part of a development team that already has significant experience with a particular web framework or programming language, then it makes sense to stick with that.
- **Productivity:** Productivity is a measure of how quickly you can create new features once you are familiar with the framework, and includes both the effort to write and maintain code (since you can't write new features while old ones are broken). Many of the factors affecting productivity are similar to those for "Effort to learn" — e.g. documentation, community, programming experience, etc. — other factors include:
  - *Framework purpose/origin*: Some web frameworks were initially created to solve certain types of problems, and remain *better* at creating web apps with similar constraints. For example, Django was created to support development of a newspaper website, so it's good for blogs and other sites that involve publishing things. By contrast, Flask is a much lighter-weight framework and is great for creating web apps running on embedded devices.
  - *Opinionated vs unopinionated*: An opinionated framework is one in which there are recommended "best" ways to solve a particular problem. Opinionated frameworks tend to be more productive when you're trying to solve common problems, because they lead you in the right direction, however they are sometimes less flexible.
  - *Batteries included vs. get it yourself*: Some web frameworks include tools/libraries that address every problem their developers can think "by default", while more lightweight frameworks expect web developers to pick and choose solution to problems from separate libraries (Django is an example of the former, while Flask is an example of a very light-weight framework). Frameworks that include everything are often easier to get started with because you already have everything you need, and the chances are that it is well integrated and well documented. However if a smaller framework has everything you (will ever) need then it can run in more constrained environments and will have a smaller and easier subset of things to learn.
  - *Whether or not the framework encourages good development practices*: For example, a framework that encourages a Model-View-Controller architecture to separate code into logical functions will result in more maintainable code than one that has no expectations on developers. Similarly, framework design can have a

large impact on how easy it is to test and re-use code.

- **Performance of the framework/programming language:** Usually "speed" is not the biggest factor in selection because even relatively slow runtimes like Python are more than "good enough" for mid-sized sites running on moderate hardware. The perceived speed benefits of another language, e.g. C++ or JavaScript, may well be offset by the costs of learning and maintenance.
- **Caching support:** As your website becomes more successful then you may find that it can no longer cope with the number of requests it is receiving as users access it. At this point you may consider adding support for caching. Caching is an optimization where you store all or part of a web response so that it does not have to be recalculated on subsequent requests. Returning a cached response is much faster than calculating one in the first place. Caching can be implemented in your code or in the server (see reverse proxy). Web frameworks will have different levels of support for defining what content can be cached.
- **Scalability:** Once your website is fantastically successful you will exhaust the benefits of caching and even reach the limits of *vertical scaling* (running your web application on more powerful hardware). At this point you may need to *scale horizontally* (share the load by distributing your site across a number of web servers and databases) or scale "geographically" because some of your customers are based a long way away from your server. The web framework you choose can make a big difference on how easy it is to scale your site.
- **Web security:** Some web frameworks provide better support for handling common web attacks. Django for example sanitises all user input from HTML templates so that user-entered JavaScript cannot be run. Other frameworks provide similar protection, but it is not always enabled by default.

There are many other possible factors, including licensing, whether or not the framework is under active development, etc.

If you're an absolute beginner at programming then you'll probably choose your framework based on "ease of learning". In addition to "ease of use" of the language itself, high quality documentation/tutorials and an active community helping new users are your most valuable resources. We've chosen Django (Python) and Express (Node/JavaScript) to write our examples later on in the course, mainly because they are easy to learn and have good support.

> **Note:** Let's go to the main websites for Django (Python) and Express (Node/JavaScript) and check out their documentation and community.
>
> 1. Navigate to the main sites (linked above)
>   - Click on the Documentation menu links (named things like "Documentation, Guide, API Reference, Getting Started", etc.)

Guide, API Reference, Getting Started", etc.).

- Can you see topics showing how to set up URL routing, templates, and databases/models?
- Are the documents clear?

2. Navigate to mailing lists for each site (accessible from Community links).
   - How many questions have been posted in the last few days
   - How many have responses?
   - Do they have an active community?

A few good web frameworks?

Let's now move on, and discuss a few specific server-side web frameworks.

The server-side frameworks below represent *a few* of the most popular available at the time of writing. All of them have everything you need to be productive — they are open source, are under active development, have enthusiastic communities creating documentation and helping users on discussion boards, and are used in large numbers of high-profile websites. There are many other great server-side frameworks that you can discover using a basic internet search.

Note: Descriptions come (partially) from the framework websites!

Django (Python)

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

Django follows the "Batteries included" philosophy and provides almost everything most developers might want to do "out of the box". Because everything is included, it all works together, follows consistent design principles, and has extensive and up-to-date documentation. It is also fast, secure, and very scalable. Being based on Python, Django code is easy to read and to maintain.

Popular sites using Django (from Django home page) include: Disqus, Instagram, Knight Foundation, MacArthur Foundation, Mozilla, National Geographic, Open Knowledge Foundation, Pinterest, Open Stack.

Flask (Python)

Flask is a microframework for Python.

While minimalist, Flask can create serious websites out of the box. It contains a development server and debugger, and includes support for Jinja2 templating, secure cookies,

unit testing , and RESTful request dispatching. It has good documentation and an active community.

Flask has become extremely popular, particularly for developers who need to provide web services on small, resource-constrained systems (e.g. running a web server on a Raspberry Pi , Drone controllers , etc.)

Express (Node.js/JavaScript)

Express is a fast, unopinionated, flexible and minimalist web framework for Node.js (node is a browserless environment for running JavaScript). It provides a robust set of features for web and mobile applications and delivers useful HTTP utility methods and middleware.

Express is extremely popular, partially because it eases the migration of client-side JavaScript web programmers into server-side development, and partially because it is resource-efficient (the underlying node environment uses lightweight multitasking within a thread rather than spawning separate processes for every new web request).

Because Express is a minimalist web framework it does not incorporate every component that you might want to use (for example, database access and support for users and sessions are provided through independent libraries). There are many excellent independent components, but sometimes it can be hard to work out which is the best for a particular purpose!

Many popular server-side and full stack frameworks (comprising both server and client-side frameworks) are based on Express, including Feathers , ItemsAPI , KeystoneJS , Kraken , LoopBack, MEAN , and Sails .

A lot of high profile companies use Express, including: Uber, Accenture, IBM, etc. (a list is provided here).

Deno (JavaScript)

Deno is a simple, modern, and secure JavaScript/TypeScript runtime and framework built on top of Chrome V8 and Rust.

Deno is powered by Tokio — a Rust-based asynchronous runtime which lets it serve web pages faster. It also has internal support for WebAssembly, which enables the compilation of binary code for use on the client-side. Deno aims to fill in some of the loop-holes in Node.js by providing a mechanism that naturally maintains better security.

Deno's features include:

- Security by default. Deno modules restrict permissions to **file**, **network**, or **environment** access unless explicitly allowed.
- TypeScript support **out-of-the-box**.
- First-class await mechanism.
- Built-in testing facility and code formatter (`deno fmt`)
- (JavaScript) Browser compatibility: Deno programs that are written completely in JavaScript excluding the `Deno` namespace (or feature test for it), should work directly in any modern browser.
- Script bundling into a single JavaScript file.

Deno provides an easy yet powerful way to use JavaScript for both client- and server-side programming.

Ruby on Rails (Ruby)

Rails (usually referred to as "Ruby on Rails") is a web framework written for the Ruby programming language.

Rails follows a very similar design philosophy to Django. Like Django it provides standard mechanisms for routing URLs, accessing data from a database, generating HTML from templates and formatting data as JSON or XML. It similarly encourages the use of design patterns like DRY ("don't repeat yourself" — write code only once if at all possible), MVC (model-view-controller) and a number of others.

There are of course many differences due to specific design decisions and the nature of the languages.

Rails has been used for high profile sites, including: Basecamp, GitHub, Shopify, Airbnb, Twitch, SoundCloud, Hulu, Zendesk, Square, Highrise.

Laravel (PHP)

Laravel is a web application framework with expressive, elegant syntax. Laravel attempts to take the pain out of development by easing common tasks used in the majority of web projects, such as:

- Simple, fast routing engine.
- Powerful dependency injection container

- Powerful dependency injection container .
- Multiple back-ends for session and cache storage.
- Expressive, intuitive database ORM .
- Database agnostic schema migrations .
- Robust background job processing .
- Real-time event broadcasting .

Laravel is accessible, yet powerful, providing tools needed for large, robust applications.

ASP.NET

ASP.NET is an open source web framework developed by Microsoft for building modern web applications and services. With ASP.NET you can quickly create web sites based on HTML, CSS, and JavaScript, scale them for use by millions of users and easily add more complex capabilities like Web APIs, forms over data, or real time communications.

One of the differentiators for ASP.NET is that it is built on the Common Language Runtime (CLR), allowing programmers to write ASP.NET code using any supported .NET language (C#, Visual Basic, etc.). Like many Microsoft products it benefits from excellent tools (often free), an active developer community, and well-written documentation.

ASP.NET is used by Microsoft, Xbox.com, Stack Overflow, and many others.

Mojolicious (Perl)

Mojolicious is a next-generation web framework for the Perl programming language.

Back in the early days of the web, many people learned Perl because of a wonderful Perl library called CGI . It was simple enough to get started without knowing much about the language and powerful enough to keep you going. Mojolicious implements this idea using bleeding edge technologies.

Some of the features provided by Mojolicious are:

- A real-time web framework, to easily grow single-file prototypes into well-structured MVC web applications.
- RESTful routes, plugins, commands, Perl-ish templates, content negotiation, session management, form validation, testing framework, static file server, CGI/PSGI detection, and first-class Unicode support.
- A full-stack HTTP and WebSocket client/server implementation with IPv6, TLS, SNI, IDNA, HTTP/SOCKS5 proxy, UNIX domain socket, Comet (long polling), keep-alive, connection pooling, timeout, cookie, multipart, and gzip compression support.

- JSON and HTML/XML parsers and generators with CSS selector support.
- Very clean, portable and object-oriented pure-Perl API with no hidden magic.
- Fresh code based upon years of experience, free and open-source.

Spring Boot (Java)

Spring Boot is one of a number of projects provided by Spring. It is a good starting point for doing server-side web development using Java.

Although definitely not the only framework based on Java it is easy to use to create stand-alone, production-grade Spring-based Applications that you can "just run". It is an opinionated view of the Spring platform and third-party libraries but allows to start with minimum fuss and configuration.

It can be used for small problems but its strength is building larger scale applications that use a cloud approach. Usually multiple applications run in parallel talking to each other, with some providing user interaction and others doing back end work (e.g. accessing databases or other services). Load balancers help to ensure redundancy and reliability or allow geolocated handling of user requests to ensure responsiveness.

Summary

This article has shown that web frameworks can make it easier to develop and maintain server-side code. It has also provided a high level overview of a few popular frameworks, and discussed criteria for choosing a web application framework. You should now have at least an idea of how to choose a web framework for your own server-side development. If not, then don't worry — later on in the course we'll give you detailed tutorials on Django and Express to give you some experience of actually working with a web framework.

For the next article in this module we'll change direction slightly and consider web security.

In this module

Last modified: Nov 17, 2021, by MDN contributors